

**AD-A253 206**



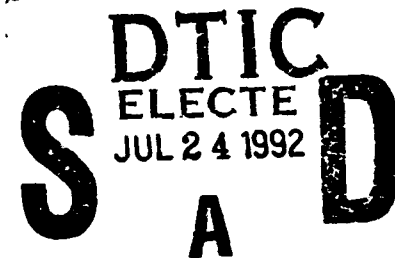
**RL-TR-92-9, Vol III (of three)  
Final Technical Report  
January 1992**



2

# **ASSURED SERVICE CONCEPTS AND MODELS Availability in Distributed MLS Systems**

**Secure Computing Technology Corporation**



**Sponsored by  
Strategic Defense Initiative Office**

*APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.*

**The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Strategic Defense Initiative Office or the U.S. Government.**

**92 7 22 031**

**Rome Laboratory  
Air Force Systems Command  
Griffiss Air Force Base, NY 13441-5700**

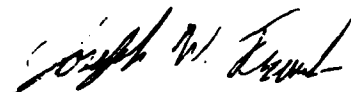
**92-19834**



This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-92-9, Vol III (of three) has been reviewed and is approved for publication.

APPROVED:



JOSEPH W. FRANK  
Project Engineer

FOR THE COMMANDER:



JOHN A. GRANIERO  
Chief Scientist  
Command, Control & Communications Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL ( C3AB), Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

ASSURED SERVICE CONCEPTS AND MODELS  
Summary

J.T. Haigh  
R.C. O'Brien  
W.T. Wood  
T.G. Fine  
M.J. Endrizzi  
S. Yalamanchili

Contractor: Secure Computing Technology Corporation  
Contract Number: F30602-90-C-0025  
Effective Date of Contract: 23 February 1990  
Contract Expiration Date: 22 February 1991  
Short Title of Work: Assured Service Concepts and Models  
Period of Work Covered: Feb 90 - Feb 91

Principal Investigator: Tom Haigh  
Phone: (612) 482-7400

RL Project Engineer: Joseph W. Frank  
Phone: (315) 330-2925

Approved for public release; distribution unlimited.

This research was supported by the Strategic Defense Initiative Office of the Department of Defense and was monitored by Joseph W. Frank, RL (C3AB) and Emilie J. Giarkiewicz, RL (C3AB) Griffiss AFB NY 13441-5700 under Contract F30602-90-C-0025.

# REPORT DOCUMENTATION PAGE

Form Approved  
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave Blank)		2. REPORT DATE January 1992		3. REPORT TYPE AND DATES COVERED Final Feb 90 - Feb 91	
4. TITLE AND SUBTITLE ASSURED SERVICE CONCEPTS AND MODELS Availability in Distributed MLS Systems				5. FUNDING NUMBERS C - F30602-90-C-0025 PE - 63223C PR - 3109 TA - 01 WU - 01	
6. AUTHOR(S) J. T. Haigh, R. C. O'Brien, W. T. Wood, T. G. Fine, M. J. Endrizzi, S. Yalamanchili					
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Secure Computing Technology Corporation 1210 West County Road, East Suite 100 Arden Hills MN 55112				8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Rome Laboratory (C3AB) Griffiss AFB NY 13441-5700				10. SPONSORING/MONITORING AGENCY REPORT NUMBER RL-TR-92-9, Vol III (of three)	
11. SUPPLEMENTARY NOTES Rome Laboratory Project Engineers: Joseph W. Frank/C3AB (315) 330-2925 Emilie J. Siarkiewicz/C3AB (315) 330-3241					
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.				12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) This report describes the work performed under the Assured Service Concepts and Models (ASCM) contract. The report is organized as follows. Volume I is a summary of all of the work done in the ASCM project. Volume II describes the various security policies that were developed on the contract. Volume III describes the availability policies that were developed on the contract and the approaches that were developed for identifying trade-offs between secrecy and availability. Volume III also contains the findings of the formalism study.					
14. SUBJECT TERMS Computer Security, Assured Service, Availability, Distributed Systems				15. NUMBER OF PAGES 88	
				16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT U/L		

## CONTENTS

Section	Page
1 Introduction	4
1.1 Purpose	4
1.2 Organization	5
2 Formalisms	6
2.1 State Machines	6
2.1.1 Advantages	7
2.1.2 Disadvantages	9
2.2 Traces	12
2.2.1 Relation with State Machines	15
2.2.2 Advantages	18
2.2.3 Disadvantages	18
2.3 Petri Nets	19
2.3.1 Relation with State Machines	21
2.3.2 Relation with Traces	21
2.3.3 Advantages	21
2.3.4 Disadvantages	22
2.4 Temporal Logic	22
2.4.1 Relation with other Formalisms	22
2.4.2 Advantages	23
2.4.3 Disadvantages	23
2.5 Interval Temporal Logic	23

Accession For	
NTIS CRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution /	
Availability	
Dist	Availability Special
A-1	

2.5.1	Relation with other Formalisms	25
2.5.2	Advantages	25
2.5.3	Disadvantages	25
2.6	Real Time Logic	25
2.6.1	Relation with other Formalisms	26
2.6.2	Advantages	26
2.6.3	Disadvantages	27
2.7	Summary	27
2.7.1	State Machines	28
2.7.2	Traces	28
2.7.3	Petri Nets	28
2.7.4	Temporal Logic	29
2.7.5	Interval Temporal Logic	29
2.7.6	Real Time Logic	29
3	Availability Models	30
3.1	A Model for Fault Tolerance	30
3.1.1	Representing Failures in the Model	31
3.1.2	A Definition of Fault Tolerance	32
3.1.3	Modeling a Fault Tolerant System	34
3.1.4	Graceful Degradation	37
3.1.5	A Comparison with other Approaches	38
3.2	Service Models	39
3.2.1	Liveness Policies	39

3.2.2	Example Real-Time Policy	53
3.2.3	Summary of the Example	69
3.3	Summary of Availability Models	70
4	Trade-offs	71
5	Conclusion	77

## SECTION 1

### INTRODUCTION

This document is a final report on research into availability models for distributed, MLS C<sup>2</sup> systems. The research has been performed under the Assured Service Concepts and Models (ASCM) project. The goals of the ASCM project are to:

- a. Develop security policies and models for distributed C<sup>2</sup> systems.
- b. Develop availability policies and models for distributed C<sup>2</sup> systems.
- c. Analyze trade-offs between security and availability.
- d. Analyze a real-system with respect to the policies and models developed on the project.

This report addresses tasks *b* and *c*.

#### 1.1 PURPOSE

In [9], computer security is defined to consist of:

- a. secrecy — protection from unauthorized disclosure of data
- b. integrity — protection from unauthorized modification of data
- c. availability — protection from unauthorized denial of Service

A system's secrecy, integrity, and availability control objectives state the secrecy, integrity, and availability requirements placed on the system.

Historical and technical reasons why denial of service has often been ignored in computer security projects are discussed in [9]. The main technical reason is that much greater progress has been made on formalizing secrecy and integrity than denial of service. Since it is difficult to analyze a system with respect to properties that are not clearly defined, it has been more feasible to analyze systems with respect to secrecy and integrity than with respect to denial of service. The author of [9] goes as far as stating:

To sum up, *security* relates to secrecy first, integrity second, and denial of service a distant third. To help you remember this, memorize the computer security researcher's favorite (tongue-in-cheek) phrase: "I don't care if it works, as long as it is secure."

The suggestion for addressing denial of service is to use structured development, fault tolerance, and software reliability techniques. These techniques are of little use for addressing denial of service unless we understand what denial of service means. So, by formalizing denial of service, we will be able to perform analysis to determine whether an application of these techniques to a system adequately addresses denial of service.



As techniques for denial of service often conflict with the secrecy control objective (see [5]), it is also important to develop techniques for analyzing trade-offs between MLS security (secrecy) and availability.

## 1.2 ORGANIZATION

In Section 2 we discuss formalisms that can be used for stating security and availability properties. In addition to describing different formalisms, this section will discuss relationships between the formalisms and advantages and disadvantages of the formalisms. In Section 3 we discuss denial of service threats and conditions. After informally describing each threat and condition, we develop a formal definition. In this section we address fault tolerance as a class of preventing denial of service. In Section 4 we discuss techniques for making trade-offs between denial of service and MLS security. Finally, in Section 5 we summarize our findings.

## SECTION 2

### FORMALISMS

Although it is possible to formalize concepts to some degree in English, inherent ambiguities in English usually prevent complete formalization. Furthermore, English is often less concise than more formal languages. Thus, before attempting to formalize concepts, it is useful to develop a formalism appropriate for the given concept. As we shall see, there does not appear to be one formalism that is appropriate for all problem domains.

As substantial work has already been devoted to developing formalisms for reasoning about computer systems, rather than developing new formalisms we describe existing formalisms that appear useful for reasoning about computer systems. Since we are using existing formalisms rather than developing new formalisms, we give only a brief description of each formalism. More in depth descriptions can be found in the cited references. After describing the formalisms, we discuss the problem domains for which each is appropriate and consider relationships that exist between the various formalisms.

We consider the following formalisms in this section:

- State Machines
- Traces
- Petri Nets
- Temporal Logic
- Interval Temporal Logic
- Real Time Logic

#### 2.1 STATE MACHINES

In this section, we describe one type of state machine model, the Mealy machine. For a more detailed description of Mealy machines as well as descriptions of other types of state machine models see [21].

A Mealy machine is characterized by the following attributes:

- a set of states,  $S$
- an initial state, *initial*
- a set of operations,  $OP$
- a set of outputs, *OUT*

- a rule indicating whether a given operation can be executed in a given state (we will use  $valid(op, st)$  to represent that  $op$  can be executed in  $st$ )
- whenever  $op$  can be executed in  $st$ , a rule indicating:
  - the state resulting from executing the operation (which we will denote by  $st^{op}$ )
  - the output that results from the operation (which we will denote by  $output(op, st)$ )

The set  $OUT$  represents the set of externally visible events. Each externally visible event is generated by some operation in the set  $OP$ . The elements of the set  $S$  provide an encoding of information about the history of the system that is needed to determine the manner in which the system should act when an operation is executed. For example:

Suppose that we consider an airplane. We might define our set of states to be:

- *PARKED* — the plane is on the ground and not moving; this is the initial state,
- *MOVING* — the plane is on the ground and moving,
- *TAKING-OFF* — the plane is in the process of taking-off,
- *LANDING* — the plane is in the process of landing,
- *FLYING* — the plane has taken off and has not yet started landing,

and our set of operations to be:

- *Start-Engine* — causes a transition from *PARKED* to *MOVING*,
- *Take-Off* — causes a transition from *MOVING* to *TAKING-OFF*
- *Land* — causes a transition from *FLYING* to *LANDING*
- *Stop-Engine* — causes a transition from *MOVING* to *PARKED*,
- *In-Air* — causes a transition from *MOVING* to *FLYING*,
- *On-Ground* — causes a transition from *FLYING* to *MOVING*

In this example, the states represent the present status of the airplane; the operations indicate the ways in which the status of the airplane can be changed; and the restrictions of the manner in which the airplane can change states in the real-world are captured in the constraints on which operations can be applied in each state.

### 2.1.1 Advantages

There are three advantages to using state machine models: they often have a clearer correspondence with the systems that they model, they allow for some automated analysis, and they specify the operations in such a manner that there is little “coupling” between operations.

Since there is no objective measure for the clarity of the correspondence between a system and a model, our claim that state machines provide a clearer correspondence is subjective. Some factors that contribute to this clarity in state machine models are:

- a. There is often a close similarity between the "data structures" used in a state machine model and those used in the actual system. For example, it seems reasonable that a flight control system would maintain the following data:

- whether the engine is on,
- whether a take-off is in process,
- whether a landing is in process,
- whether the plane is airborne

each of these data components can be represented by a Boolean variable resulting in a state comprised of four Boolean variables. Then, the state of the actual system can be represented as follows:

$$\langle V_1, V_2, V_3, V_4 \rangle$$

where  $V_1$ - $V_4$  are Boolean values indicating the value of each of the four Boolean variables identified above. Then, the mapping between the state machine model and the actual system is:

- *PARKED*  $\leftrightarrow \langle F, F, F, F \rangle$ ,
- *MOVING*  $\leftrightarrow \langle T, F, F, F \rangle$ ,
- *TAKING-OFF*  $\leftrightarrow \langle T, T, F, F \rangle$ ,
- *FLYING*  $\leftrightarrow \langle T, F, F, T \rangle$ ,
- *LANDING*  $\leftrightarrow \langle T, F, T, T \rangle$

The close correspondence between states in the model and states in the actual system facilitates developing an accurate model of the system and drawing conclusions about the actual system from analysis of the model.

Note that it is not generally true that all of the states in the actual system correspond to states in the model. For example,  $\langle F, F, F, T \rangle$  does not correspond to any of the states in our example model. This is the state in which the flight control system believes that the plane is airborne even though the engine is off. This is not valid in the simple model presented here (although it could be addressed by adding a state, such as *STALLED*, to the model). In order to establish the consistency between our example model and actual system, it is necessary to address the possibility of invalid states being reached in the actual system.

- b. State machines provide a procedural view of the world.

As implementation languages are often procedural, a modeling approach that emphasizes the procedural aspects of computing is often appropriate.

We do not mean to suggest that state machine models are always more understandable than models in other formalisms. Care must be taken when developing the model to choose states and operations that are easily mapped to the states and operations of the actual system.

If the number of states and operations in a state machine model is small, then it is often possible to automate the analysis. For example, it is often desirable to determine whether certain states are reachable from other states. This question can be answered by performing a brute force analysis to determine all the states reachable from a given initial state. Clearly this analysis becomes infeasible if there are too many states or operations. This does not mean that it is not possible to analyze systems with a large number of states, it just means that a more intelligent analysis procedure must be used.

A common approach that is taken with safety problems is to model the system as a state machine, determine the set of states that are "safe", and show that it is not possible to effect a transition from a "safe" state to an "unsafe" state as the result of executing an operation. This provides an inductive argument that all the reachable states are "safe" whenever the initial state is safe.

By saying that the specifications of the operations are loosely "coupled", we mean that it is possible to understand how any given operation behaves without understanding how the other operations behave. This makes the specifications easier to understand because it is often possible to understand how a given operation behaves without understanding how the whole system works. Maintenance of the specifications is also simplified as there is less danger that a change in the specification of an operation will ripple through the specifications of many other operations.

### 2.1.2 Disadvantages

There are three drawbacks to using the state machine paradigm: there are times in which it is desirable to make distinctions between different types of operations, it is difficult to represent concurrency with state machines, and state machines often have more implementation detail than is necessary.

In the state machine model for the airplane, the operations can be divided into two classes. The first class, *{Start-Engine, Stop-Engine, Land, Take-Off}*, consists of operations that more naturally fit the intuitive notion of "executing". For example, it is natural to have an operation corresponding to the pilot starting the engine. The second class, *{In-Air, On-Ground}*, consists of operations that seem artificial. For example, there is not really an *In-Air* operation that can be executed to cause the plane to become airborne. As a result of the physical laws governing the behavior of the plane, it becomes airborne at some time after the initiation of the take-off. As we will see later, other formalisms allow us to consider becoming airborne and touching down simply as events that occur rather than as operations.

Since each operation in a state machine model is atomic, it is difficult to represent concurrency in a realistic manner using state machines. For example:

Suppose we consider a database system. For simplicity, we will consider the case in which there is only one data item in the system. We might model the system as having one state corresponding to each possible value for the data item and operations *Update*<sub>v</sub> to change the value associated with the data item to *v* and *Retrieve* to retrieve the value associated with the data item.

Suppose that our values are pairs of integers  $(v_1, v_2)$  and that we want to enforce  $v_1 > v_2$  as a consistency constraint. To analyze whether the system enforces this constraint, we might take the following steps:

- a. Check that  $v_1 > v_2$  in the initial state.
- b. Check that the *Update* operation only changes the state when  $v_1 > v_2$  and then it does so by transitioning to the state indicated by  $(v_1, v_2)$ .

Although this is a nice, simple model of the system and it is easy to determine whether the consistency constraint is enforced, the model and analysis are worthless if the system allows for concurrent update operations to be executed. For example, if *Update*<sub>(2,1)</sub> and *Update*<sub>(300,4)</sub> are executed concurrently, the following might occur in the system:

- a. Since 300 is greater than 4, the system allows *Update*<sub>(300,4)</sub> to change the state.
- b. As a first step in changing the state, the system changes the first component of the state to 300.
- c. Since 2 is greater than 1, the system allows *Update*<sub>(2,1)</sub> to change the state.
- d. The system changes the first component of the state to 2 and the second component to 1.
- e. The system completes the *Update*<sub>(300,4)</sub> operation by setting the second component of the state to 4.

So, the final state reached in the system will be (2, 4) which violates the consistency constraint.

In order to address this issue, the model must be detailed enough to represent the concurrency in the system. For example, we might change our model as follows:

Let  $B_1$  and  $B_2$  be bags<sup>1</sup> of integers and  $v_1$  and  $v_2$  be integers. Then, we can define our states to be tuples  $\langle B_1, B_2, v_1, v_2 \rangle$ . Intuitively,  $B_1$  and  $B_2$  would represent values that have been requested for  $v_1$  and  $v_2$ , respectively.

In addition to the *Update*<sub>v</sub> operations, we would have operations *Update1*<sub>i</sub> and *Update2*<sub>i</sub> for each integer *i*. The state transition function would be defined so that *Update*<sub>v</sub> adds  $v_1$  and  $v_2$  to  $B_1$  and  $B_2$ , respectively, if  $(v_1, v_2)$  is a valid request. The operation *Update1*<sub>i</sub> can only be executed when *i* is an element of  $B_1$  in the current state. In that case, a transition is made to a state in which:

---

<sup>1</sup> A bag is a generalization of a set in which the number of occurrences of each element is significant in addition to which elements are present.

- $B_1$  has one occurrence of  $i$  removed.
- $v_1$  has value  $i$ .
- $B_2$  and  $v_2$  are unchanged.

$Update2_i$  is defined in a similar fashion.

Then, the model is sufficiently detailed for the violation of the consistency constraint to be visible. For example:

Suppose the initial state is  $\langle \{\}, \{\}, a_1, a_2 \rangle$  and we consider the operation sequence  $Update_{(300,4)}$ ,  $Update_{(2,1)}$ ,  $Update1_{300}$ ,  $Update1_2$ ,  $Update2_1$ ,  $Update2_4$ . Then, the following states would be reached:

- a.  $\langle \{300\}, \{4\}, a_1, a_2 \rangle$ ,
- b.  $\langle \{2, 300\}, \{1, 4\}, a_1, a_2 \rangle$ ,
- c.  $\langle \{2\}, \{1, 4\}, 300, a_2 \rangle$ ,
- d.  $\langle \{\}, \{1, 4\}, 2, a_2 \rangle$ ,
- e.  $\langle \{\}, \{4\}, 2, 1 \rangle$ ,
- f.  $\langle \{\}, \{\}, 2, 4 \rangle$ .

Clearly, the final state violates the consistency constraint. Depending on the values of  $a_1$  and  $a_2$ , it is possible that some of the intermediate states also violate the consistency constraint.

Once the flaw in the system has been detected and some means of correcting it has been determined, the model can be updated to reflect the new design of the system and the analysis can be repeated to determine whether the new system design is appropriate.

The above example demonstrates that rather than being impossible to address concurrency in a state machine model, it is only more difficult. While it is generally true that concurrency is more difficult to analyze, the state machine paradigm seems ill-equipped to address the issue. For example, the operations  $Update1_i$  and  $Update2_i$  are artificial in the sense that they are not operations that a user explicitly executes but rather portions of the processing required to respond to an explicit user request.

Since it is typically necessary to consider nondeterminism to adequately address concurrency, a further complication of the use of our state machine model to address concurrency is that it does not allow nondeterministic operators. Although [21] describes how nondeterministic operators can be described in the framework of deterministic state machines, the deterministic state machine models obtained generally have too many states for analysis to be feasible. Once again, the analysis is theoretically possible, yet very difficult in the state machine paradigm.

In the previous section we claimed that one of the advantages of the state machine paradigm was that there often was a close correspondence between the “data structures” in a state machine model of a system and the data structures in the actual system. To many, this represents a serious disadvantage of the state machine paradigm because the model might have more implementation detail than it would in other paradigms. As we will see in the following sections, there is a strong connection between all of the formalisms. Thus, although it might be common for users of the state machine paradigm to include more implementation detail than is necessary, it is quite often the case that a more carefully built model would be just as abstract as models in other formalisms. In other words, there is actually nothing inherent in the paradigm that requires implementation detail in the model.

## 2.2 TRACES

While the state machine paradigm concentrates on the current state of the system, the trace paradigm concentrates on the allowable execution histories of the system. In other words, while the state machine paradigm relies on the system states to capture the aspects of the execution history of the system that are relevant to describe future behavior of the system, the trace paradigm allows explicit reference to be made to the execution history. The particular instance of the trace paradigm that we will discuss in this section is CSP as described in [13].

CSP provides two mechanisms for the describing the behavior of a system: the process language and *sat* predicates. Underlying both of these mechanisms are *events*.

Events in the CSP model of a system are used to mark points in time at which system relevant activities occur. Referring to our airplane example, we might have events:

- *start* — request to start engine,
- *stop* — request to stop engine,
- *to* — request to initiate take-off,
- *l* — request to initiate landing,
- *ab* — detection that plane has become airborne,
- *td* — detection that plane has touched down

The set of all of the events used in modeling the system is called the *alphabet* of the system.

Many aspects of the behavior of the system can be described by specifying which sequences of events from the alphabet correspond to allowable behaviors of the system. For example, assuming that the plane starts in a state in which the engine is off, the plane may not become airborne until the engine is started. This means that *start* must occur before *ab*. Thus, the sequence  $\langle ab \rangle$  is not a valid execution history of the system. The set of valid execution histories of the system is called the *traces* of the system.



In [13], two classes of systems are described. The first class consists of all systems that can be described solely in terms of its traces. These are referred to as the *deterministic processes*. All other systems are termed *nondeterministic processes*. [13] characterizes the nondeterminism in terms of two constructs, the *failures* and *divergences* of the system. Intuitively, the failures describe actions in which the system is ready to participate but chooses to ignore while the divergences describe instances in which the system reaches a state from which it is impossible to reason about its behavior. It is beyond the scope of this report to completely address the failures and divergences. In this section, we will only consider deterministic processes. For an in-depth discussion of the nondeterministic processes, see [13].

In order to specify a system using a *sat* predicate, a Boolean expression is constructed that captures the relevant constraints on the behavior of the system. Then, it is asserted that the system *satisfies* the Boolean expression. A system's *sat* predicate is simply the Boolean expression that the system is asserted to satisfy. For example, we might use the following as the *sat* predicate for the airplane example:

$$\begin{aligned} tr_{i+1} = start &\Rightarrow (i = 1 \text{ or } tr_i = stop) \\ \text{and } tr_{i+1} = stop &\Rightarrow (tr_i = start \text{ or } tr_i = td) \\ \text{and } tr_{i+1} = to &\Rightarrow (tr_i = start \text{ or } tr_i = td) \\ \text{and } tr_{i+1} = l &\Rightarrow tr_i = ab \\ \text{and } tr_{i+1} = ab &\Rightarrow tr_i = to \\ \text{and } tr_{i+1} = td &\Rightarrow tr_i = td \end{aligned}$$

where *tr* represents an arbitrary trace of the system and  $tr_i$  denotes the  $i^{th}$  event in *tr*.

In this case, we have derived the *sat* predicate by simply defining for each event the set of events that can precede it. Since the *sat* predicate can be any Boolean expression defined on traces, this provides a very powerful means for specifying behavior.

The drawback of using *sat* predicates to define behavior is that it is difficult to determine their consistency and completeness. When a *sat* predicate is used to specify system behavior, it provides an implicit definition of the set of traces. Our above specification of the behavior of the airplane should be read as asserting that the set of traces is comprised of exactly those sequences that satisfy the *sat* predicate. If the *sat* predicate is too weak, then our specification will be incomplete and we will be allowing some behaviors that we do not mean to allow. For example, if we remove the second conjunct (the requirement that  $tr_{i+1} = stop \Rightarrow (tr_i = start \text{ or } tr_i = td)$ ), then we have no constraint on when the *stop* event can occur. Thus, we would be allowing system behaviors in which the engine stopped in mid-air. On the other hand, if the *sat* predicate is too strong, then our specification will be inconsistent and will disallow certain behaviors that we would like to allow. For example, if the second conjunct were strengthened to require that the *stop* event only occur directly after the *start* event, then we would not allow the engine to be stopped after the plane touches down. As an even more extreme example, if the *sat* predicate is logically inconsistent it will not be satisfied by any event sequences and the set of traces will be empty.

To facilitate the specification of the set of traces for a system, CSP provides the process language. The process language provides an algebra for building processes. For example, given a process  $P$  and an event  $e$ , CSP uses:

- $e \rightarrow P$  to denote the process that first does  $e$  and then behaves like the process  $P$ , and
- $(e_1 \rightarrow P_1) \parallel (e_2 \rightarrow P_2)$  to denote the process that either does  $e_1$  and then behaves like  $P_1$  or does  $e_2$  and then behaves like  $P_2$ .

Using these operators from the process language, we can specify the airplane as the process  $P$  where:

- $P = \text{start} \rightarrow M$   
All a parked plane can do is starts its engine and behave like a moving plane.
- $M = (to \rightarrow F) \parallel (stop \rightarrow P)$   
A moving plane either takes-off and behaves like a flying plane or stops its engine and behaves like a parked plane.
- $F = ab \rightarrow l \rightarrow td \rightarrow M$   
A flying plane becomes airborne, starts landing, touches down, and then acts like a moving plane.

Given an operator  $OP$  in the process language and processes  $P_1, \dots, P_n$  for which the sets of traces are already known, [13] provides rules for determining the set of traces for the process built from  $P_1, \dots, P_n$  using  $OP$ . These rules allow us to obtain the traces for a complicated system by determining the traces for a set of simple systems and combining the simple systems in an appropriate fashion.

The approach used in [13] is to use the process language to specify the behavior of the system and to use *sat* predicates to state properties of the system that are of interest. With this approach, the processes  $P$ ,  $M$ , and  $F$  above would be the definition of the system while the *sat* predicate that we earlier used as the definition of the system would simply be a consequence of the process language specification of the system. The only time that a *sat* predicate should be used to define the behavior of the system rather than calling out a consequence of the behavior specified by the process language is when a choice has been made to not specify the system in the process language.

The following constructs described in [13] will be used in this report:

- $P \parallel Q$  — parallel composition of processes  $P$  and  $Q$  with interaction between  $P$  and  $Q$ ,
- $!e \rightarrow P$  — the process that outputs event  $e$  and then behaves like process  $P$ ,
- $?x \rightarrow P$  — the process that inputs a value for  $x$  and then behaves like process  $P$ ,
- $\mu X.F(X)$  — the process  $X$  with  $X = F(X)$ ; this allows us to recursively define processes,
- $P/s$  — the process that behaves like  $P$  behaves after partaking in the trace  $s$ ,

- $P|||Q$  — the parallel composition of processes  $P$  and  $Q$  with no interaction between  $P$  and  $Q$ ,
- $P//Q$  — the process that behaves like the parallel composition of processes  $P$  and  $Q$  with events for  $P$ 's alphabet made invisible

We will also use the following notation from [13]:

- $\langle e_1, e_2, \dots, e_n \rangle$  — the sequence  $e_1, e_2, \dots, e_n$ ;  $\langle \rangle$  denotes the empty sequence,
- $s \downarrow e$  — the number of occurrences of event  $e$  in sequence  $s$ ,
- $s \hat{\sim} t$  — the catenation of sequences  $s$  and  $t$ ,
- $\surd$  — a special event denoting successful termination; for example, to indicate that a process completes successfully after partaking in the events in a sequence  $s$ , we say  $s \hat{\sim} \langle \surd \rangle$  is a trace of a process,
- $s|A$  — the sequence  $s$  with all elements that are not in  $A$  removed,
- $\#s$  — the length of the sequence  $s$ ,
- $P^0$  — the set of events in which process  $P$  is willing to partake;  $e \in P^0$  if and only if  $\langle e \rangle$  is a trace of  $P$ ,
- $s_0$  — the first element of a sequence  $s$ ,
- $\bar{s}$  — the reverse of the sequence  $s$ ; note that  $\bar{s}_0$  is the last element of  $s$ ,
- $SKIP$  — a special process that is a no-op other than successfully completing
- $x : B \rightarrow P(x)$  — a shorthand for:

$$x_1 \rightarrow P(x_1) | \dots | x_r \rightarrow P(x_r)$$

where  $B$  is a set of  $r$  events and each  $P(x_i)$  is the process to be executed after the occurrence of  $x_i$ .

Complete definitions of the constructs and notation can be found in [13].

### 2.2.1 Relation with State Machines

Before discussing the advantages and disadvantages of using CSP, we will examine the relationship between CSP models and state machine models. First we will show that it is easy to translate a state machine model into CSP.

Let the states of the model be  $st_1, st_2, \dots, st_n$  and the operations be  $op_1, op_2, \dots, op_m$ . Suppose that

$$B_k = \{op_{k_1}, \dots, op_{k_{j(k)}}\}$$

is the set of operations that can be executed from  $st_k$  and that  $st_{k,i}$  is the resulting state when operation  $i$  is applied to  $st_k$ . For each  $i$  define a CSP event  $OP_i$  to represent the execution of operation  $i$ . Then, define a CSP process  $ST_k$  for each  $st_k$  as follows:

$$ST_k = OP_i : B_k \rightarrow ST_{k,i}$$

If  $st_k$  is the initial state of the system, then the process  $P = ST_k$  is an equivalent formulation of the system in terms of CSP. For example, our CSP model of the airplane can be derived automatically from our state machine model of the airplane by using the process derived above.

Now, we will consider translating a CSP model into a state machine model. Suppose that  $P$  is a CSP process which models the system in question. For each trace  $t$ , we define:

$$ST_t = \{s \in \text{traces}(P) : P/s = P/t\}$$

Recall that given a process  $P$  and a trace  $s$ , the expression  $P/s$  in CSP denotes the process that behaves like  $P$  behaves after participating in the events in  $s$ .

So,  $ST_t$  contains all of the traces of  $P$  for which  $P$ 's future behavior is the same as  $P$ 's future behavior after participating in  $t$ .

Given  $ST_t$  and an event  $e$  of  $P$ 's alphabet, consider whether  $t \cdot \langle e \rangle$  is a trace. If it is, then define:

$$\text{valid}(e, ST_t) = \text{TRUE}$$

$$ST_t^e = ST_{t \cdot \langle e \rangle}$$

otherwise, define

$$\text{valid}(e, ST_t) = \text{FALSE}$$

It is straightforward to show that this provides a state transition function that is well-defined on the set of states. The state machine model obtained is equivalent with the CSP model in the sense that:

- Any trace of the CSP model is a valid execution sequence for the state machine model and vice-versa.
- The behavior exhibited by the CSP model after participating in the events in a trace  $t$  is the same as that of the state machine model when started in the state  $ST_t$ .

Applying this procedure to the CSP model of the airplane results in states:

- $ST_{\langle \rangle}$
- $ST_{\langle \text{start} \rangle}$

- $ST_{\langle start, to \rangle}$
- $ST_{\langle start, to, ab \rangle}$
- $ST_{\langle start, to, ab, l \rangle}$

with the following transitions possible:

- $start^{ST} \langle \rangle = ST_{\langle start \rangle}$
- $stop^{ST} \langle start \rangle = ST_{\langle \rangle}$
- $to^{ST} \langle start \rangle = ST_{\langle start, to \rangle}$
- $ab^{ST} \langle start, to \rangle = ST_{\langle start, to, ab \rangle}$
- $l^{ST} \langle start, to, ab \rangle = ST_{\langle start, to, ab, l \rangle}$
- $td^{ST} \langle start, to, ab, l \rangle = ST_{\langle start \rangle}$

By equating the states and operations as follows:

- $ST_{\langle \rangle} \leftrightarrow PARKED$
- $ST_{\langle start \rangle} \leftrightarrow MOVING$
- $ST_{\langle start, to \rangle} \leftrightarrow TAKING-OFF$
- $ST_{\langle start, to, ab \rangle} \leftrightarrow FLYING$
- $ST_{\langle start, to, ab, l \rangle} \leftrightarrow LANDING$
- $start \leftrightarrow Start-Engine$
- $stop \leftrightarrow Stop-Engine$
- $to \leftrightarrow Take-Off$
- $ab \leftrightarrow In-Air$
- $l \leftrightarrow Land$

- $td \leftrightarrow \text{On-Ground}$

it is clear that this is equivalent to our state machine model of the airplane.

Thus, we see that it is straightforward to translate between deterministic processes in CSP and state machines. Further research needs to be done to determine whether the above construction is suitable for nondeterministic processes.

### 2.2.2 Advantages

Although CSP processes can be translated into state machines, there are certain advantages to using CSP. Some of the major reasons are that CSP supports reasoning about concurrency, CSP facilitates the development of abstract system models, and CSP does not make any distinction between a process and its environment.

Concurrency can be represented in a natural way by the parallel composition operator ( $\parallel$ ) in CSP. For example,  $P \parallel Q$  represents  $P$  operating concurrently with  $Q$  with synchronization occurring on events common to both processes. As there are defined operators and associated proof rules for concurrency, CSP provides direct support for reasoning about concurrent processing.

Rather than considering a systems internal state, CSP considers events that are visible external to the system. This makes it possible to construct models that make no reference to the internal structure of the system. Thus, it is easy to construct models that abstract away implementation details.

CSP considers each process' environment to be a process itself. Thus, CSP can be used to specify both a process and the environment of the process. This makes it possible to make assertions about and reason about the environment of a process.

### 2.2.3 Disadvantages

A conscious decision was made in the development of CSP to ignore the concept of causality. A consequence of this is that there is no way to specify that a process causes an event to occur. Processes have no control over when events occur, they merely partake in processes that occur. In certain parts of [13], it seems that the environment is responsible for generating events. As mentioned earlier, the environment of a process is a process itself; thus, it does not seem consistent to view the environment as producing events.

In general, various aspects of CSP seem unintuitive. This means that the formal manipulations performed in the analysis of a CSP model can violate the intuition of the analyst unless the analyst is *very* experienced in the use of CSP. This is discouraging in that it results in a steep learning curve for CSP and makes it difficult to determine the correctness of CSP models.

Another potential problem with using CSP is that the standard mode of defining processes is by using the process language. This often results in the system being specified as the composition

(using a variety of the CSP operators) of a relatively large set of processes. This can make it very difficult to understand the system's behavior from the specification.

In summary, the main disadvantage of CSP seems to be that it is such a powerful language that a thorough understanding of CSP is required to use it. On the other hand, a formalism such as state machines is fairly understandable even to novices.

## 2.3 PETRI NETS

As with the other formalisms, there are several variations of Petri nets. In this section, we will only consider the simplest version of Petri nets. For a more complete description of Petri nets, see [19].

A Petri net model of a system contains four components:

- $\mathcal{P}$  — the set of places,
- $\mathcal{T}$  — the set of transitions,
- $\mathcal{I}$  — a mapping from transitions to bags of places,
- $\mathcal{O}$  — a mapping from transitions to bags of places,

Each place is a container for tokens. An assignment of tokens to places is called a *marking*. Thus, a marking is a mapping from places to nonnegative integers. Suppose that for transition  $t$ , place  $p$  occurs  $n$  times in the bag of places specified by  $\mathcal{I}(t)$ . Then, if  $\mu$  is a marking with  $\mu(p) \geq n$ , then we say that  $t$  can fire in  $\mu$ . When  $t$  fires in  $\mu$ , tokens are removed from each  $p$  in  $\mathcal{I}(t)$ ; the number of tokens removed is equal to the number of times that  $p$  occurs in  $\mathcal{I}(t)$ . Similarly, tokens are added to each  $p$  in  $\mathcal{O}(t)$ ; one token is added to  $p$  for each occurrence of  $p$  in  $\mathcal{O}(t)$ .

To see how Petri nets can be used to model machines in a natural fashion, suppose we view a machine as being defined by *events*<sup>2</sup> and *conditions*. An event is an action that can occur in the system and a condition is a predicate. We can view the state of the system as being defined by a set of predicates. Each event can be described in terms of the conditions that must hold for it to occur, its *preconditions*, and the conditions that it causes to hold in the system, its *postconditions*. Then, a Petri Net representation of the system can be constructed by:

- a. Defining a place for each condition in the system.
- b. Defining a transition  $t$  for each event  $e$  in the system so that:

- (1) If  $p$  is a place that corresponds to a precondition of  $e$ , then  $p \in \mathcal{I}(t)$ .

---

<sup>2</sup>In this section, "event" has a different connotation than it does in the discussion of CSP.

(2) If  $p$  is a place that corresponds to a postcondition of  $e$ , then  $p \in \mathcal{O}(t)$ .

For example, consider our airplane example. The events are the system operations *Start-Engine*, *Take-Off*, *Land*, *Stop-Engine*, *In-Air*, and *Ground*. We define  $t_{sta}$ ,  $t_{to}$ ,  $t_l$ ,  $t_{sto}$ ,  $t_{ia}$ , and  $t_g$  to be the transitions corresponding to these events.

The conditions are the predicates we defined in Section 2.1.1:

- the engine is on
- a take-off is in progress
- a landing is in progress
- the plane is airborne
- the engine is off
- a take-off is not in progress
- a landing is not in progress
- the plane is not airborne

We define  $p_o$ ,  $p_t$ ,  $p_l$ ,  $p_a$ ,  $p_{no}$ ,  $p_{nt}$ ,  $p_{nl}$ , and  $p_{na}$  to be the places corresponding to these conditions. The reason it is necessary to have the predicates  $p_{no}$ ,  $p_{nt}$ ,  $p_{nl}$ , and  $p_{na}$  is that the variety of Petri Net that we are using in this section does not allow a transition to test whether a place is empty. So, although it is clear that we intend "the engine is off" to correspond to  $p_o$  being empty, we must explicitly add a place representing "the engine is off" for our transitions to make use of this condition.

The preconditions are simply the constraints on when an operation can be applied. For example, since the plane must be airborne before landing,  $p_a$  is a precondition for  $t_l$ . The postconditions are derived from the state changes caused by the transition. For example, since the engine enters the on state after it is turned on,  $p_o$  is a postcondition for  $t_{sta}$ .

The complete definition of  $\mathcal{I}$  and  $\mathcal{O}$  is:

$t$	$\mathcal{I}(t)$	$\mathcal{O}(t)$
$t_{sta}$	$\{p_{no}\}$	$\{p_o\}$
$t_{to}$	$\{p_o, p_{nt}, p_{nl}, p_{na}\}$	$\{p_o, p_t, p_{nl}, p_{na}\}$
$t_l$	$\{p_o, p_a, p_{nl}\}$	$\{p_o, p_{na}, p_l\}$
$t_{sto}$	$\{p_o, p_{na}, p_{nl}, p_{nt}\}$	$\{p_{no}, p_{na}, p_{nl}, p_{nt}\}$
$t_{ia}$	$\{p_t, p_{na}\}$	$\{p_{nt}, p_a\}$
$t_g$	$\{p_l\}$	$\{p_{nl}\}$



### 2.3.1 Relation with State Machines

Each marking can be considered a state and each transition can be considered an operation. Then, the state transition function can be defined from  $\mathcal{I}$  and  $\mathcal{O}$  from the firing rules defined above. Thus, it is straightforward to map Petri nets into state machines.

Although a state machine can be translated into a Petri net by the following steps:

- a. Define one place for each state.
- b. Given  $st$  and  $op$  such that  $op$  can be executed in  $st$  and results in  $st_2$ , define a transition  $t_{st,op}$  that removes a token from the place corresponding to  $st$  and adds a token to the place corresponding to  $st_2$ .
- c. Require that the initial marking of the system always have all of the places empty except for one which contains a single token

this transformation would not be very useful. In the worst case, it would result in the number of transitions in the Petri net being equal to the product of the number of states and the number of operations in the state machine model. As the places can only represent integer values, it is not clear how a state machine can automatically be mapped into a Petri net in a useful way.

### 2.3.2 Relation with Traces

It is also possible to think of the firing of a transition as being an event. Then, a Petri net can be translated into a deterministic CSP process as follows:

Say that  $s$  is a firing sequence from marking  $\mu$  in a Petri net if:

$s$  consists of a single transition  $t$  that can fire in  $\mu$ , the initial marking of the system,  
or  $s = \langle t \rangle \hat{\ } s_1$  where  $t$  can fire in  $\mu$  and  $s_1$  is a firing sequence for  $\mu_t$ , the marking resulting from  $t$  firing in  $\mu$

In other words,  $s$  is a possible execution history of the Petri net. So, the traces of the CSP process can be defined to be the set of firing sequences from  $\mu_0$ , the initial marking of the Petri net.

As with state machines, although it is possible to automatically construct a Petri net from a deterministic CSP process, it is difficult to do so in a useful manner.

### 2.3.3 Advantages

Petri nets are useful for simulating systems due to the simple execution rules. They can also be used to analyze safety properties automatically through *reachability trees* (see [19]) as long as the number of transitions is not too large. Thus, Petri nets share some of the advantages of state machines.

As Petri nets do not specify the order of firing when more than one transition can fire in a given marking, they can be used to represent concurrency. Thus, they share some of the advantages of CSP.

### 2.3.4 Disadvantages

The class of Petri nets that we have described provides little support for reasoning about data flow in a system. For example, it would be difficult to model the reading and writing of data values using Petri nets. They are better suited for reasoning about the control flow in a system. For example, in the model of the readers/writers problem in [19], the values of the data being read and written are ignored. All that is considered is whether the data is being read while it is being written. Thus, the flavor of Petri nets described in this section is not applicable when data values are relevant.

## 2.4 TEMPORAL LOGIC

Although there are several flavors of temporal logic, we will only consider the basic flavor in this section. Models are constructed by combining predicates using logical operators. In addition to the standard logical operators, two additional operators are provided.

- a.  $\Box P$  — denotes that predicate  $P$  is satisfied for all future times,
- b.  $\Diamond P$  — denotes that predicate  $P$  is satisfied for some future time

These operators can be combined to form more complicated expressions. For example:

- $\Diamond \Box P$

denotes that there is some time after which  $P$  is always true while

- $\Box \Diamond P$

denotes that it is always the case that at some later time  $P$  will be true.

### 2.4.1 Relation with other Formalisms

In general it is difficult to relate models written in temporal logic with models written in formalisms such as state machines, CSP, and Petri nets since there is very little structure to a temporal logic specification. The other formalisms provide an explicit model while temporal logic provides more of an implicit model by making assertions about the behavior of the system.

### 2.4.2 Advantages

The main advantage of temporal logic is that it allows for reasoning about temporal properties. This is not all that significant since it is possible to perform similar reasoning in other formalisms (see Section 3.2.1).

### 2.4.3 Disadvantages

The main disadvantage of temporal logic is that it is a very primitive specification language. Unlike state machines, CSP, and Petri nets, there is no concept of processing inherent in temporal logic. This means that the developer of the model is responsible for building concepts from scratch. Also, it is nontrivial to determine the completeness and consistency of a temporal logic specification (just as it is nontrivial to determine the completeness and consistency of a process specified by a *sat* predicate in CSP).

It is also important to note that while temporal logic can be used to specify temporal properties, it cannot be used to state real-time policies. Thus, temporal logic falls very short of totally addressing time. This failure is partially addressed by interval temporal logics which are discussed in the next section.

## 2.5 INTERVAL TEMPORAL LOGIC

Interval temporal logics (ITLs) are flavors of temporal logic in which operators are provided for specifying intervals of time. Since many interesting properties are easily expressed using intervals of time, ITL specifications are often nicer than temporal logic specifications. For example, we can assert that an airplane remains airborne between when it takes off and when it lands as follows:

During the interval of time between when the plane becomes airborne and when the plane lands, it is always the case that the plane is airborne.

It is difficult to express assertions about bounded time intervals in temporal logic since  $\Box$  and  $\Diamond$  do not place any restrictions on how far into the future they extend. To address this issue, some flavors of temporal logic include a third temporal operator, *Until* that can be used to bound time intervals. ITL is based on the assumption that intervals are a more natural way to specify restrictions on the scope of  $\Box$  and  $\Diamond$ . There are two flavors of ITL. In this report, we will consider the flavor of ITL described in [17].

Assertions in ITL are of the form:

$$[I]\alpha$$

where

- $I$  denotes a time interval, and

- $\alpha$  is a temporal logic assertion

For example:

- $[I]P$  asserts that  $P$  is true at the first time in interval  $I$ ,
- $[I]\Box P$  asserts that  $P$  is true at every time in interval  $I$ ,
- $[I]\Diamond P$  asserts that  $P$  is true at some time in interval  $I$

Thus, if  $I$  is an interval that begins with the airplane becoming airborne and ends just prior to the airplane landing and  $A$  denotes that the airplane is airborne, then:

$[I]\Box A$

asserts that the airplane is airborne between when it takes off and when it lands.

ITL provides the following constructs:

- Given a predicate  $P$ ,  $[P]$  denotes an interval that begins with  $P$  becoming true,
- $*I$  denotes that interval  $I$  exists,
- *before*  $I$  denotes the time just prior to the beginning of  $I$ ,
- $I \Rightarrow J$  denotes the interval that starts at the end of  $I$  and ends at the end of the next  $J$ ,
- $I \Leftarrow J$  denotes the interval that ends at the end of  $J$  and begins at the end of the first  $I$  preceding  $J$

For example, if  $L$  denotes that the airplane has landed and  $T$  denotes that the airplane is taking off, then we can formalize the assertion that the airplane is airborne between when it takes off and when it lands as:

- $[(T \Rightarrow A) \Rightarrow \text{before } L]A$

[17] also defines the following constructs to address real-time:

- $I \Rightarrow I + t$  denotes an interval of length  $t$  starting at the beginning of interval  $I$ ,
- $I - t \Rightarrow I$  denotes an interval of length  $t$  ending at the beginning of interval  $I$ ,
- $[I] < t$  denotes that the length of interval  $I$  is less than  $t$ ,

- $[I] > t$  denotes that the length of interval  $I$  is greater than  $t$

For example,

- $[A \Rightarrow \text{before}T] < 10\text{hours}$

asserts that an airplane must always land within 10 hours of when it becomes airborne.

### 2.5.1 Relation with other Formalisms

Since ITL is a flavor of temporal logic the discussion in Section 2.4.1 is also relevant to ITL.

### 2.5.2 Advantages

The only advantage that ITL has over other formalisms is that it allows for the statement of temporal and real-time properties. Although many other formalisms can be used to state temporal properties, there are few formalisms that can be used to state real-time properties. Also, as many interesting temporal properties are statements about intervals in time, ITL sometimes allows for a more natural specification than might be possible in other formalisms.

### 2.5.3 Disadvantages

As with temporal logic, there is no inherent notion of processing in ITL. So, ITL does not provide much support for the specification of processes either.

Another disadvantage is that no proof rules are provided for ITL in [17]. Instead it is suggested that a decision process be used to automatically analyze ITL specifications. Although [17] claims that ITL is decidable, it is not clear whether an efficient decision procedure exists for ITL. If the decision process is not efficient, then it is not reasonable to perform the analysis automatically and it is necessary to have proof rules for the logic.

If there is an efficient decision process, then the fact that the analysis can be done automatically would be an advantage. If not, then it is necessary to determine whether useful proof rules can be derived for ITL before using it to model systems.

## 2.6 REAL TIME LOGIC

Real Time Logic (RTL) is described in [6]. Rather than having the  $\Box$  and  $\Diamond$  operators it makes use of an explicit notion of time and expressions quantified over time. For example,  $\Box P$  can be represented as:

$\forall t, P(t)$

RTL is consistent with CSP in that it contains the notions of events and processes. RTL distinguishes between the following classes of events:

- Start Events —  $\uparrow P$  denotes the start of execution of process  $P$ ,
- Stop Events —  $\downarrow P$  denotes the completion of execution of process  $P$ ,
- Transition Events —  $(S := T)$  denotes the changing of value of predicate  $S$  to true while  $(S := F)$  denotes the changing of value of predicate  $S$  to false
- External Events — events that cannot be caused to happen by the system

RTL uses the @ operator to define the time of the  $i^{th}$  occurrence of an event. Given an event  $e$  and a positive integer  $i$ :

- $@(e, i)$

denotes the time at which the  $i^{th}$  occurrence of  $e$  occurs. If we view  $A$  as a state predicate that denotes whether the airplane is airborne and  $G$  as a state predicate that denotes whether the airplane is on the ground, the assertion:

$$\forall i, t \text{ such that } @(A := T), i \leq t < @(G := T), i + 1 : A(t)$$

specifies that the airplane remains airborne between when it takes off and when it lands. The reason that we are interested in occurrence  $i+1$  of  $(G := T)$  rather than occurrence  $i$  is that we assume that the plane is initially on the ground. Thus,  $(G := T)$  is assumed to happen at time 0. Consequently, the  $i^{th}$  occurrence of  $(G := T)$  will always precede the  $i^{th}$  occurrence of  $(A := T)$ .

### 2.6.1 Relation with other Formalisms

RTL seems to combine aspects of various formalisms. Since RTL contains the notion of state predicates, it is possible to use RTL in such a way that it is similar to the state machine paradigm. Similarly, since RTL contains the notion of processes and events, it can be used in ways that are similar to CSP. There are also similarities between RTL and ITL. For example, ITL treats the changing of a predicate's value as an event just as RTL does.

### 2.6.2 Advantages

Since RTL incorporates aspects of many other formalisms, it shares most of the advantages of the other formalisms. For example, it addresses concurrency and real-time and provides models that are understandable yet free from implementation detail.

### 2.6.3 Disadvantages

If real-time is not a concern, then there is no advantage to using RTL instead of CSP. In fact, there is an advantage in using CSP in that the process language for CSP is much richer than that for RTL.

There are often times when it is difficult to say simple things in RTL. For example, whereas ITL allows us to refer to the interval of time between when the plane becomes airborne and the first subsequent time the plane lands, RTL forces us to know which occurrence of  $(G := T)$  occurs next. From the design of our system we happened to know that occurrence  $i + 1$  of  $G := T$  is the first occurrence subsequent to  $@((A := T), i)$ , there are times when it is very difficult to determine which occurrence occurs next. A related issue is that if we changed our system to require that the airplane be airborne at time 0, then our assertion would have to be changed to:

$$\forall i, t \text{ such that } @((A := T), i) \leq t < @((G := T), i) : A(t)$$

This can result in the specifications being difficult to maintain. For example, a change in an initial condition might result in many changes. It also requires information to be included in the problem that is not necessarily relevant to the problem. For example, is it necessary to know whether the airplane was initially on the ground in order to determine whether it is always airborne between when it takes off and when it lands.

## 2.7 SUMMARY

In this section we summarize the results of our formalism study by discussing the problem domains to which each formalism is appropriate. Because of the strong connections between the various formalisms, it is usually the case that anything that can be done using one formalism can be done using an appropriate variety of any other formalism. So, in the following we consider a formalism to be appropriate for a problem domain only when it can be used in a *straightforward* way to obtain useful results about the problem domain.

Our formalism study found CSP to be the most reasonable formalism for  $C^2$  systems when real-time is not an issue. When real-time is an issue, CSP does not appear to be appropriate. Then, a formalism such as ITL or RTL is needed. It is not clear that either of ITL and RTL is superior to the other. Our specific findings about each formalism are summarized in the following table.

Formalism	Most Appropriate Problem Domain
State Machines	Stand-alone transaction processing systems
Traces	Distributed systems which interact with their environment
Petri Nets	Policies with control flow more important than the data flow
Temporal Logic	Systems with temporal policies
Interval Temporal Logic	Systems with real-time policies
Real Time Logic	Systems with real-time policies

### 2.7.1 State Machines

As the state machine paradigm does not address concurrency very well, it is more suited to stand-alone systems than it is to distributed systems. As the transitions represent operations, the state machine paradigm most naturally fits with systems that accept inputs and provide responses to those inputs.

Thus, state machines are most appropriate to stand-alone, transaction processing systems. They are least appropriate for distributed systems that, in addition to accepting input, actively request information from their environment. For example, in our model of an airplane, it is unnatural to view the event when the airplane becomes airborne as an operation.

As there is a strong connection between addressing real-time and addressing concurrency, state machines do not appear to be appropriate for reasoning about real-time.

### 2.7.2 Traces

As CSP has a steeper learning curve than state machines, it is less appropriate for the stand-alone, transaction processing systems that state machines address adequately. This does not mean that CSP cannot be used for such systems, it just means that CSP might be overkill in these cases.

Of the formalisms we examined, CSP seems the most appropriate for distributed systems that actively query the state of the environment. As these are defining characteristics of  $C^2$  systems, we feel that CSP is the most appropriate formalism to use in the portions of this report that ignore real-time issues. The variety of CSP defined in [13] does not address real-time and thus is not appropriate for systems in which real-time issues are important.

We have studied proposals for adding real-time to CSP and investigated adding real-time to CSP ourselves and reached the conclusion that it is very difficult to extend CSP to address real-time. The main cause of the problem is that since events simply occur rather than being generated by anything, it is difficult to define semantics that allow for the specification that an event must occur at a specific time while remaining consistent with the semantics already defined for CSP.

### 2.7.3 Petri Nets

Since Petri nets do not address data flow very well, they are only useful when control flow is more important than data flow in the system policy. Since Petri nets can be translated into CSP and CSP addresses control flow very nicely, we see no reason to use Petri nets for formalizing security and denial of service concepts. If it is desirable to perform a simulation of the system, then it might be reasonable to construct a Petri net model of the system for simulation purposes. The Petri net model could then be translated into CSP for the formal analysis.



#### **2.7.4 Temporal Logic**

Since temporal properties can be stated in other formalisms and temporal logic provides very little structure for constructing system models, we see no reason to use temporal logic for formalizing security and denial of service concepts. Instead, it seems more reasonable to derive operators in the other formalisms that correspond to the temporal operators in temporal logic. For example, in Section 3.2.1 we discuss defining the concept of eventuality in the CSP formalism.

#### **2.7.5 Interval Temporal Logic**

Since temporal properties can be addressed in other formalisms, ITL appears to only be appropriate for addressing real-time systems.

#### **2.7.6 Real Time Logic**

Since RTL does not provide as rich of a process language as CSP, there is no point in using RTL unless real-time issues are relevant. Although RTL can be used to address real-time systems, there are times that it might result in unwieldy specifications due to, for example, the need to explicitly specify which occurrence of an event is of concern rather than simply referring to the next occurrence. In these cases, ITL is more appropriate to use than RTL. Otherwise, it is more appropriate to use RTL than ITL because it is more readable.

## SECTION 3

### AVAILABILITY MODELS

In this section we describe availability models for  $C^2$  systems. First, we formalize the notion of fault tolerance. Next, we formalize certain classes of service policies. In analyzing a system, we would first use the fault tolerance policy developed in Section 3.1 to determine whether the system behaved as specified even in the presence of specified classes of faults. Then, we would analyze the system specifications with regard to the service policies developed in Section 3.2. This analysis will determine whether the system satisfies the service policies even in the presence of the specified classes of faults.

#### 3.1 A MODEL FOR FAULT TOLERANCE

The following terminology is used throughout this section:

- A *failure* is a deviation between the specified behavior of an entity and its observed behavior.
- A *fault* is the phenomenological cause of a failure.

Our model of fault tolerance is based on the description found in [2]. The entire system is viewed as a hierarchy of subsystems. Each subsystem represents a *server* that provides a service to a *client* (or *user*). To provide this service, the subsystem must, in turn, request service from lower level subsystems, which may be software or hardware components of the system. These lower level subsystems are called *resources* for the server, and the server is said to *depend* on those resources that it directly invokes. A particular subsystem can be thought of as either a client, server or resource depending on the level at which the subsystem is being viewed. Hardware subsystems can also be viewed as part of this hierarchy, and, in general, appear in the hierarchy at the lower levels. Figure 3-1 represents one level of the hierarchy. The arrows in the figure represent requests and responses between clients and the server and, at the next lower level where the server is the client, between the server and its resources.

Failures may occur at any level within any subsystem, either hardware or software, of the hierarchy. A classification of the types of failures that might occur is given in section 3.1.1. When a resource that a server depends upon experiences a failure whose consequences are propagated to the server, then a fault has occurred for the server. The manner in which the server handles the fault and the number and types of faults that the server can adequately handle determine the degree to which the server is fault tolerant. Faults that a server cannot adequately handle are propagated upward to the server's clients as failures of the server, of a possibly different type.

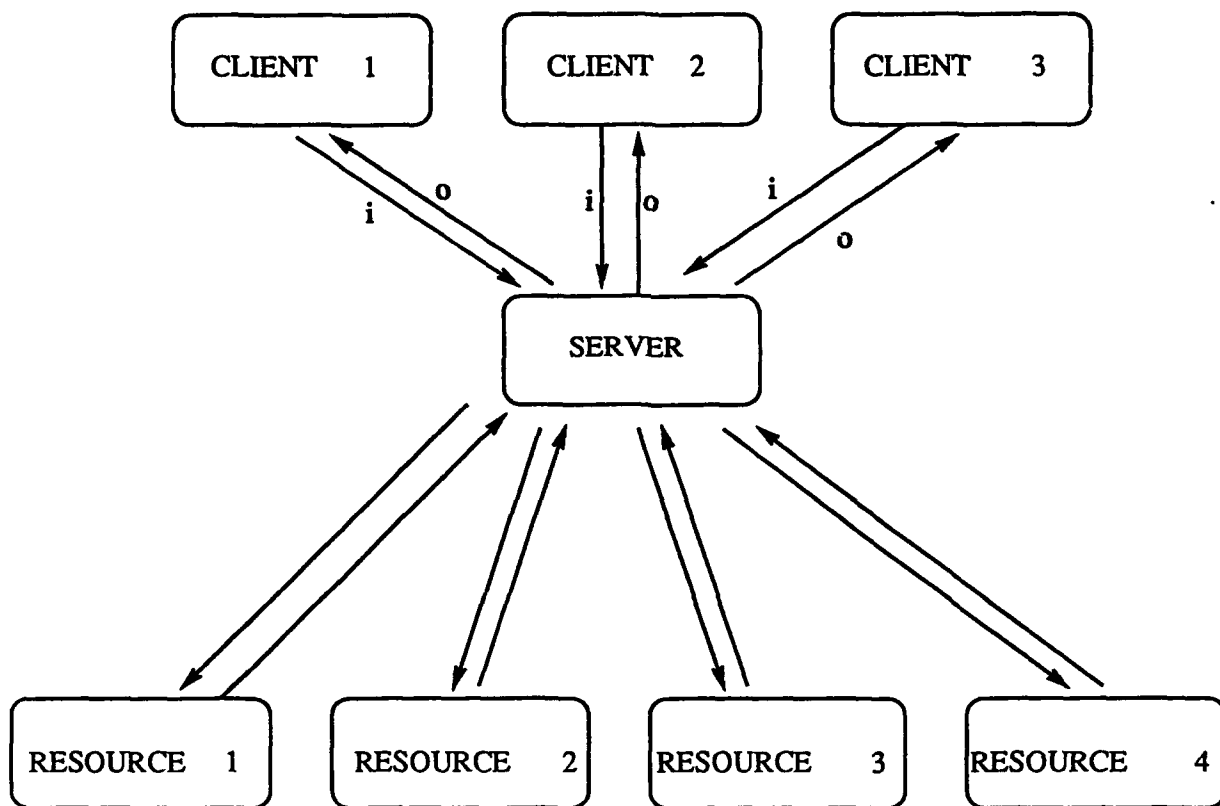


Figure 3-1. The System Server Model

### 3.1.1 Representing Failures in the Model

A server's specifications define the *correct* operation of the server in response to an input. *Failures* correspond to incorrect behavior by the server in response to an input. In [2] a classification of types of failures that a server can experience is given. This section contains a summary of this classification. Specific examples of each type of failure are given in section 3.1.3.

A server's specification is assumed to define both the server's response to a given input in a given state and the time interval during which the response should occur. The response includes both the output that is returned to the client and any state changes that result.

**Response failures** occur when the response of the server is not correct. There are two ways in which the response may fail to be correct.

- A **value failure** occurs if the output returned to the client is not correct.
- A **state transition failure** occurs if the state does not change in the manner specified.

Note that a response failure may involve both a value failure and a state transition failure.

**Timing failures** occur when the response is correct but does not take place within the time interval specified. This could happen either because

- the response is too **early**, or
- the response is too **late** - a **performance failure**.

**Omission failures** occur when the server fails to respond at all to an input.

**Crash failures** occur when the server no longer responds to any input until the system is restarted.

Crash failures can be classified depending on the manner in which the system is restarted.

- A **halting crash** occurs when the system cannot be restarted.
- A **total amnesia crash** occurs if the server must be restarted in a predefined initial state that is not dependent on any inputs that occurred before the crash.
- A **partial amnesia crash** occurs if the server is restarted in a state that is a combination of portions of the state before the crash and portions of a predefined initial state.
- A **pause crash** occurs if the server is restarted in the state prior to the crash.

### 3.1.2 A Definition of Fault Tolerance

In order to state a formal definition of fault tolerance, we use the trace semantics of Hoare's CSP [13]. In this formalism, the behavior of a system is completely specified by its set of possible traces. The definition can then be stated in terms of conditions on the set of traces. Since the definition deals with faults, we need to specifically identify which events correspond to faults and which correspond to the requests and responses between the server and its clients.

A system (server) is defined by a tuple  $(E, F, I, O, Tr)$  where

- $E$  represents the set of possible events in the system.
- $F$  represents the subset of  $E$  that consists of all those events which are faults.
- $I$  represents the subset of  $E$  that consists of all those events that are "requests for service" from the system.
- $O$  represents the subset of  $E$  that consists of all those events that are responses from the system to requests for service.

- $Tr$  represents the set of traces of the system.  $Tr$  is a subset of  $E^*$  and defines the possible behavior of the system.

In terms of figure 3-1, if the system is the server, then  $E$  represents the possible events that the server can perform and  $Tr$  the set of sequences of events.  $I$  represents possible requests that the clients can make on the server and  $O$  the possible responses to these requests that the server can make. The set  $E$  includes the events  $I$ ,  $O$ , events internal to the server, and events describing the interaction between the server and lower level resources that it depends on. These events are subsets of the input and output events of the lower level servers. The set  $F$  must include all of the faults that are to be addressed in the analysis. For example, if the analysis is meant to address hardware failures, then  $F$  would be defined to contain those responses that the server receives from lower level services that can be classified as failures.

The fault tolerance of the server can be measured in terms of the degree to which it can tolerate faults from its lower level resources and still return the appropriate responses to its clients. In order to analyze the behavior of the server in the presence of different faults, we will consider traces on the system that contain these faults. Such traces are called fault scenarios. The intuition behind this approach is that if the server is fault tolerant, then the observable behavior of the server in the trace after the fault appears should be no different from the observable behavior of the server if the fault did not occur.

Following [22] we define a fault scenario  $C$  as a subset of  $E^*$ . We make the additional restriction that  $C$  be closed under the prefix operation<sup>3</sup>. This means that  $C$  describes a subset of the traces of the system. We use  $N$  to denote the set of all traces of the system that have no faults in them; that is,  $N = Tr \cap (E - F)^*$ . Since we are only concerned with the manner in which the system handles faults, we will assume that all fault scenarios  $C$  discussed have  $N$  as a subset. Fault scenarios may be defined in a number of ways. For example,  $C$  may be the set of all traces that have at most one fault, or the set of all traces whose only faults are omission faults.

**Definition:** A server system  $S$  is fault tolerant with respect to a given fault scenario  $C$  if the behavior of  $S$  as observed by its clients is unaffected by any faults that occur in  $C$ .

More formally,

**Definition 3.1** If  $\beta \in C$ , then  $\exists \gamma \in N \ni \beta|IO = \gamma|IO$ , where  $IO = I \cup O$  and  $|$  denotes the restriction operator.

While the faults that the resources may generate cover all classes of faults described in section 3.1.1 including timing faults, it should be noted that our trace model has no way of specifying exact time. In particular, if a response time requirement exists for the server, this definition has no way of addressing whether or not that requirement is satisfied. The definition does address temporal order, however, and if, in the presence of a fault, the server's responses are identical but occur in a different sequence, this behavior is identified by the definition as behavior that is not fault tolerant.

<sup>3</sup>A set  $S$  of sequences is said to be *closed under the prefix operation*, or *prefix closed*, if  $t \in S$  whenever  $s \in S$  and  $t$  is an initial subsequence of  $s$

Adding response time requirements to the definition would involve using a temporal model and had some notion of causality. This is an area for future research. Another area for future research is the investigation of fault tolerance for systems that are nondeterministic in the sense of [13]. Since our definition only refers to the traces of the system and the nondeterministic processes cannot be characterized solely in terms of traces, it is clear that our definition does not address the nondeterministic processes.

Note also that this definition does not capture a stochastically recognizable change in behavior of the system that may occur in the presence of nondeterminism. For example, suppose that given a certain sequence of inputs, if there is no fault, then output  $o_1$  is observed with probability  $p$  and  $o_2$  with probability  $1 - p$ . If a fault occurs, then output  $o_1$  is observed with probability  $p'$  and  $o_2$  with probability  $1 - p'$  where  $p$  and  $p'$  are sufficiently different that the change is noticeable. This type of change in behavior is allowed by the above definition of fault tolerance. While such a change in behavior is of concern when the security of the system is analyzed, it would only be of concern in a fault analysis of the system if the probabilities of the various outputs are part of the system's specifications. By changing the probabilities at which the outputs occur, it may be possible that the system's specifications are no longer satisfied. This amounts to the lower level fault resulting in the server displaying a different type of failure. Developing a model that includes stochastic behavior is beyond the scope of the current contract, but is an interesting area for future research.

Our definition of fault tolerance is a refinement of that given in [22] and [3]. In those papers fault tolerance is defined as follows:

**Definition 3.2**  $\forall \beta \in Tr, \beta \in C \Rightarrow \beta|\bar{F} \in Tr$ , where  $\bar{F}$  denotes the complement of  $F$  in  $E$ .

The main differences between this definition and Definition 3.1 are that Definition 3.1 explicitly identifies those events that are involved in determining whether a server is fault tolerant, and it also allows internal events, that may take place in the server when recovering from faults, to be part of the model. The following example illustrates this point.

**Example:** In this example we use  $e_i$  to denote events that are internal to the system but not failures and  $f_i$  to denote internal events that are failures.

Suppose that  $\gamma = \langle i_1 e_1 e_2 o_1 \rangle$  is in  $Tr$  and is the behavior of the system, given input  $i_1$ , when a fault does not occur. Suppose that  $\beta = \langle i_1 e_1 f_1 e_3 e_4 o_1 \rangle$  is in  $C$  and is the behavior of the system when a fault does occur. As far as the client is concerned, the system tolerates the fault  $f_1$  since for an input of  $i_1$ , an output of  $o_1$  is received regardless of whether or not the fault occurs.

Then  $\gamma|IO = \beta|IO$  so the definition of fault tolerance given in Definition 3.1 is not violated.

However,  $\beta|\bar{F} = \langle i_1 e_1 e_3 e_4 o_1 \rangle$ , which is not a trace, so Definition 3.2 is not satisfied.

### 3.1.3 Modeling a Fault Tolerant System

In this section we discuss an example, based on Example 1 in [12], and show how the various concepts and definitions apply to it. This example involves a cab company in which customers

telephone their requests into dispatchers who assigned priorities to the requests and placed them into a priority queue for servicing. When drivers become available, they are dispatched to pick up the customer in the queue with the highest priority. In order to increase the dependability of the service, the priority queue is replicated at several sites throughout the city.

In our discussion of the example, we will use the same scenario. However, it should be apparent that the example could easily be rephrased in a  $C^2$  setting.

In terms of the server model for fault tolerance, the clients are the customers, the inputs to the system are the customer requests for service and the outputs are the dispatching of drivers to service the request. The system itself is a logical entity that relies on the information available from each site and driver. So these sites and drivers represent the resources that the system depends upon. The internal events involve keeping track of the status of the drivers, polling the various sites for information and sending information to them.

The service is fault tolerant if each request is serviced by exactly one driver in the order that is determined by the priority of the request. The internal events include the actions taken within the dispatching system to assign priorities to requests, to insert the requests properly into the priority queue and to maintain the queue in a consistent manner across the replicated sites.

Failures that might occur include individual sites crashing or a failure in communication among the sites causing the replicated queues to become inconsistent.

The system performs two operations in managing the customer queue: enqueueing a request, when a call from a customer arrives, and dequeuing a request when a driver is dispatched to service a request. Both the enqueue and dequeue requests require communication between the sites to maintain the queue in the appropriate manner. The sites that are notified of an enqueue request constitute the final enqueue quorum for the request; when a driver is available, the sites that are queried to determine which customer should be picked up constitute the initial dequeue quorum; and the sites that are notified when a driver is dispatched to pick up a customer constitute the final dequeue quorum. In order to guarantee that the queue is maintained as a one-copy serializable version of a replicated priority queue, it has been shown [11] that the following conditions must hold:

- a. Each initial dequeue quorum intersects each final enqueue quorum.
- b. Each initial dequeue quorum intersects each final dequeue quorum.

Now assume that the cab system operates as follows. When a customer places a call to a particular site, that site enqueues the request in its version of the priority queue and broadcasts a message to all the other sites with the information. The sites that receive the enqueue message update their priority queues accordingly, and these sites constitute the final enqueue quorum. If there are no failures, the final enqueue quorum should be all of the sites. Similarly when a particular site is notified that a driver is available to be dispatched, then it dispatches the driver to the first customer in its priority queue and then broadcasts this information to all the other sites. The sites that receive the dequeue message update their priority queues accordingly, and these sites

constitute the final dequeue quorum. The initial dequeue quorum for each site is just the site itself since no other sites were queried as to which customer to service.

(We assume that the broadcasts require a fixed amount of time and that after broadcasting the dequeue information, the site waits to make sure that no dequeue request is received from another site for the same customer. If such a dequeue request is received, then some procedure exists for determining which site makes the dispatch.)

The particular classes of failures can appear in the system in the following way when requests are made of the resources.

- A broadcast message that is garbled upon receipt represents a value failure.
- An error at a site in recording a correctly received broadcast message from another site represents a state transition failure.
- A broadcast message that takes longer than the allocated amount of time represents a performance failure. (There are no cases of early timing failures in this example.)
- Inability to broadcast a message from a site represents an omission failure.
- The complete failure of a site or several sites represents a crash failure.
- If all of the site's current state is destroyed as a result of the crash, including its version of the priority queue and any knowledge of requests received or dispatches made by the site prior to the crash, then this represents a total amnesia crash.
- If a site's priority queue can be restored on restart, but the most recent request or dispatch is lost, then this represents a partial amnesia crash.
- A failure in the communications system at a site represents a pause crash since the state of the site will be the unchanged when communications are restored.

If no failures occur, then conditions  $a$  and  $b$  will be satisfied and the customer's requests will be serviced properly. Suppose, however, that a site does not receive an enqueue broadcast from another site. The effect would be that this request would not get enqueued in the priority queue of other site. As a consequence, the other site would never dispatch a driver to service the request represented by the enqueue broadcast and this request might get serviced out of its priority order. Similarly, if the site did not receive a dequeue broadcast from another site, then it would not know that the particular request was already serviced. In this case it might dispatch another driver to service the request.

In both cases, the system is not fault tolerant, and, as expected, does not satisfy our definition of fault tolerance. To see this, we need to consider possible traces of the system. Suppose that

- $i_{x,a}$  represents the initial request  $x$  by a customer to site  $a$ ;
- $ib_{x,ab}$  represents the successful broadcast of request  $x$  from site  $a$  to site  $b$ , for  $b \neq a$ ;
- $if_{x,ab}$  represents the failed broadcast of request  $x$  from site  $a$  to site  $b$ , for  $b \neq a$ ;



- $o_{x,a}$  represents the dispatching of a driver to handle request  $x$  by a given site  $a$ ;
- $ob_{x,ab}$  represents the successful broadcast of a dispatch for request  $x$  from site  $a$  to site  $b$ , for  $b \neq a$ ;
- $of_{x,ab}$  represents the failed broadcast of a dispatch for request  $x$  from site  $a$  to site  $b$ , for  $b \neq a$ .

Then  $\beta = \langle i_{x,1} \text{ } ib_{x,12} \text{ } ib_{x,13} \text{ } o_{x,2} \text{ } ob_{x,21} \text{ } of_{x,23} \text{ } o_{x,3} \rangle$  is a possible trace of the system in which two drivers are dispatched to service the same request. This behavior occurs because the broadcast message that site 2 sent to site 3 indicating that site 2 had already sent a driver to handle request  $x$  was not received by site 3. Hence, site 3 thinks that the request  $x$  has still not been handled and dispatches a driver. In this case,  $\beta|IO = \langle i_{x,1} \text{ } o_{x,2} \text{ } o_{x,3} \rangle$  which is not equal to  $\gamma|IO$  for any  $\gamma \in N$ .

Similarly, if the priority given to a request is based on servicing the requests in the order received, then  $\beta = \langle i_{x,1} \text{ } ib_{x,12} \text{ } if_{x,13} \text{ } i_{y,3} \text{ } ib_{y,31} \text{ } if_{y,32} \text{ } o_{y,3} \rangle$  is a possible trace of the system in which the requests are serviced out of order. This occurs because the broadcast message that site 1 sent to site 3 indicating that request  $x$  had been received was not received by site 3. Hence, when request  $y$  arrives, site 3 believes that this is the most recent request and dispatches a driver to handle it. In this case,  $\beta|IO = \langle i_{x,1} \text{ } i_{y,3} \text{ } o_{y,3} \rangle$  which is not equal to  $\gamma|IO$  for any  $\gamma \in N$ .

One way of adding fault tolerance to the system would be to enlarge the initial dequeue quorum. This could be done by requiring a site to request information on who to dequeue from  $2n + 1$  other sites, and then acting on the majority response. Such a system could tolerate  $n$  broadcast failures of the type in the previous examples. An additional complication occurs, however, because of the possibility that the messages that are sent in order to determine an initial dequeue quorum might also fail. A possible approach might be for a site to send  $2n + 1$  messages requesting information and, after a certain period of time, if a majority of responses that agreed with one another had not been received, to resend messages to the sites that had not yet responded. The resending of messages could be done as often as needed. In this manner, the system could be made fault tolerant for faults that arise as the result of trying to determine an initial dequeue quorum. If we include the events that are part of determining an initial dequeue quorum in the model, then we have an example in which faults may result in different internal events occurring but the ultimate response from the system being the same. This is just the type of situation that can be handled by Definition 3.1 but not by Definition 3.2.

### 3.1.4 Graceful Degradation

Once a definition of fault tolerance has been decided upon, then it is possible to define various versions of graceful degradation by weakening the definition of fault tolerance in appropriate ways. This has been done in [3] and, in a slightly different way, in [12].

The approach taken in [12] is based on defining fault tolerance using constraints on the system. If all of the constraints are satisfied, then the system is fault tolerant. As more and more of the constraints are not satisfied, the behavior of the system is said to degrade. A structure can be imposed on the fault tolerant behavior of the system by considering the set of all possible subsets of constraints, i.e. the power set of the set of constraints. This set forms a lattice under set

inclusion. By mapping system behavior to the set of constraints that are satisfied by the system, it is possible to place a partial order on the different behaviors of the system. This partial order can then be used to describe the manner in which the system gracefully degrades.

The approach taken in [3] is based on two possible ways in which the definition of fault tolerance could be weakened. This approach extends easily to our definition of fault tolerance.

- A smaller set of faults might be tolerated. In the definition, this involves letting the set  $C$  of fault scenarios which the system tolerates become progressively smaller. This amounts to saying that, as the system degrades, there are more and more types of faults from which the system can not shelter the client.
- Another form of graceful degradation might involve the manner in which the system reacts to a fault. It may be the case that the system cannot totally shelter the client from the occurrence of faults but that it can control the effect that these faults have on the system behavior in a well-defined way. In the definition, for the same set  $C$ , rather than requiring that  $\beta|IO = \gamma|IO$ , it might only be required that  $\beta|IO \approx \gamma|IO$  where  $\approx$  denotes some relation on sequences that is weaker than equality. For example, the order of output events may be permuted, or equality may only be required to hold on a subset of the output events. Or for a model involving time, the response time to a request might fall outside a specified range.

### 3.1.5 A Comparison with other Approaches

The main appeal of Definition 3.1 as a definition of fault tolerance is that it is conceptually very simple and clear. It is given at a high enough level that system details do not show up in the definition. This allows one to focus on exactly what is meant by fault tolerance independent of a particular system.

The definition also allows one to distinguish between whether the system is fault tolerant as opposed to whether it is "correct", i.e. satisfies its specifications. The reason for this is that the definition is not based on any requirements or specifications for the system. In particular, it only compares the observable behavior of the system in the presence of faults with the behavior in the absence of faults. It says nothing about whether the behavior of the system in the absence of faults is correct.

While this distinction may be viewed as an advantage, there are a number of limitations to the definition and the trace model used to describe it. Two of these have been mentioned earlier: the fact that timing specifications, and therefore timing failures, cannot be represented in this model, and the fact that stochastic behavior also cannot be represented in the model. Both of these limitations are a consequence of the definition not depending on the specification of system behavior at the client's level. That is, there is nothing in the definition that talks about system failures, only failures at the resource level. In order to extend the definition to cover all cases, it will probably be necessary to define fault tolerance in terms of the specifications of the system at the client's level. The definition of fault tolerance can then be rephrased in terms of the specifications as follows.

**Definition:** A server system  $S$  is fault tolerant with respect to a given fault scenario  $C$  if the specifications of the system  $S$  are satisfied for all system behaviors contained in  $C$ .

More formally,

**Definition 3.3** *If  $P$  is the process defined by the traces in  $C$  and  $SPEC$  is the set of specifications for  $S$ , then  $P \text{ sat } SPEC$ ; i.e. the specifications are satisfied by each trace in  $C$ .*

This approach is the more traditional approach to fault tolerance found in [12] and others. Graceful degradation comes from relaxing the set  $SPEC$  of specifications that must be satisfied. This could be done by changing the specifications or by just requiring that only some of them be satisfied. The latter approach is the one taken by [12] described earlier. It should also be pointed out that if the trace model used supports real-time specifications or specifications of stochastic behavior, then the same definition can still be used to include timing failures and stochastic misbehavior.

## 3.2 SERVICE MODELS

In this section we describe classes of availability policies that are applicable to  $C^2$  systems. First, we describe policies that do not require an explicit notion of time. We then provide a worked example of a real-time policy.

### 3.2.1 Liveness Policies

A liveness property of a system is a temporal property that "something desirable eventually happens" as the system evolves. For example, successful termination of a sequential program is a liveness property. We are concerned here with modes of service denial in which process interaction denies a liveness property to some process that was designed to possess it. The next section introduces a notational extension to CSP that will facilitate the formalization of these service denial modes, the following section defines and formalizes deadlock, starvation, and fairness, and the last section discusses associated service policies.

**3.2.1.1 Notation** — We first introduce a mild extension of CSP, *behaviors*, to facilitate the definition of liveness properties and situations that deny liveness. The extension is mild because it applies CSP theory and does not add anything to the theory. Later we introduce the basic notion of *temporal dependence* which lies at the heart of the modes of service denial discussed here. In addition to the notation defined in this section, we also make use of the notation defined in Section 2.2.

The trace model of the evolution of a CSP process is strongly biased towards looking backwards in time from some point in time "now" towards the beginning of time, i.e. the start of the process. It is easy to refer to a trace  $t$  that contains an occurrence of an event  $e$ , as in " $e$  in  $t$ ". If a trace  $t$  of a process  $P$  ends with, say, the  $n^{\text{th}}$  occurrence of event  $e$ , then  $t$  is the concatenation of the trace  $s$

of a particular evolution of  $P$  with  $e$ :  $t = s^{\cdot}(e)$ . The trace  $s$  records a history of the behavior of  $P$  prior to the  $n^{\text{th}}$  occurrence of  $e$ , and is available with no extra work. The trace model makes it very easy to assert safety properties of a process, which correspond to "always" assertions in temporal logic: A safety property is an assertion that is true of all traces of a process. It should be noted that a safety property is an *invariant* of a process.

Liveness properties, which correspond to "sometimes" or "eventually" assertions in temporal logic, are another matter. The closest we can get to expressing the idea that "Event  $e$  eventually occurs in the lifetime of process  $P$ " with the basic notational devices of CSP is to say that there is an  $N$  such that all traces longer than  $N$  contain an occurrence of  $e$ . The difficulty with this is, of course, that such a global bound on the number of elapsed events until occurrence of the event in question may not be known, and may not even exist. In order to express liveness properties of a CSP process, we will construct a new process attribute called the process' *behaviors*.

The construction is similar to that in the theory of finite automata [14] by which the next-state function  $\delta$  is extended to a function of history  $\hat{\delta}$ . Given a CSP process we build a possibly infinite set of sequences of events, any of which could be infinitely long; each sequence is one possible lifetime behavior of the process. For generality we start with the theory of nondeterministic processes laid out in [13], pp. 129 - 132 and briefly described in Section 2.2. By using this model rather than the trace model used previously in this report, it is possible to define liveness policies for nondeterministic processes.

For the remainder of this section let  $P = \langle A, F, D \rangle$  be a process. Also, given a finite set  $A$  we use  $A^{\infty}$  to denote the set of countably infinite sequences of elements of  $A$  and  $A^{\omega}$  to denote the set of finite or countably infinite sequences of elements of  $A$ . Clearly  $A^{\omega} = A^* \cup A^{\infty}$ . The connection between this model of processes and the model for deterministic processes used earlier is:

- $A$  is the set of events for the process.
- $F$  is a set of pairs  $(s, X)$  where  $s \in A^*$  and  $X \subseteq A$
- $D$  is a set of elements from  $A^*$
- The set of traces of the process consists of all  $s$  such that there exists some  $X$  with  $(s, X) \in F$ .

So, the model of process used in this section is an extension of the model used earlier in that in addition to capturing the traces of the process it captures the following information:

- $(s, X) \in F$  denotes that after participating in the events in  $s$  the process can refuse to participate in any further events from  $X$
- $s \in D$  denotes that after participating in the events in  $s$  the process can forevermore choose to participate in or refuse to participate in any event

The set  $F$  is included to allow a distinction to be made between a process that must participate in an event and a process that may choose to participate in an event. The traces only capture the events in which a process may participate; they do not consider whether the process has any choice to refuse to participate in an event. For example, suppose that  $P_1$  is a server that is required to

process service requests from other processes and  $P_2$  is the same as  $P_1$  except it may internally choose to refuse certain requests. Since both processes can participate in the same event sequences, they have the same traces. The set  $F$  distinguishes between the two processes by recording that  $P_2$  may refuse certain events that  $P_1$  must accept. For a more detailed description of this model and the differences between deterministic and nondeterministic processes, see [13].

A finite behavior represents the sequence of events a process participates in until it stops, due either to failure or to successful termination. A trace of  $P$  after which  $P$  has a non-empty refusal set is a behavior that terminates, due either to failure or to success:

**Definition 3.4**  $t \in A^*$  is a *terminating behavior* of  $P$  if and only if there is a pair  $\langle t, B \rangle \in F$  such that  $B \neq \emptyset$ .

If a trace of  $P$  has  $\sqrt{\phantom{x}}$  as its last element it is a behavior that terminates successfully. CSP uses the symbol " $\sqrt{\phantom{x}}$ " as a special marker event to indicate successful completion of a terminating sequential process; see [13], pp. 171ff.

An infinite behavior represents the sequence of events a nonterminating process participates in throughout its lifetime. In the following, for any  $x, y \in A^\omega$  we use  $x < y$  when  $x$  is a proper initial segment of  $y$ , i.e.

**Definition 3.5** For any  $x, y \in A^\omega$ ,  $x < y$  if and only if there is a  $z \in A^\omega$  such that  $z \neq \langle \rangle$  and  $y = x\hat{\phantom{x}}z$ .

When  $y = x\hat{\phantom{x}}z$  we say that  $z$  is the *complement in  $y$  of  $x$* . (Note for future reference: It can be shown that for any  $x, y \in A^\omega$ ,  $x < y$  only if  $x \in A^*$ ).

Every finite initial segment of an infinite behavior of a process is a trace of the process, and the following event is one of the events the process may participate in after the segment:

**Definition 3.6**  $h \in A^\omega$  is a *nonterminating behavior* of  $P$  if and only if for each subsequence  $s < h$  there is an event  $e \in A$  such that  $\langle e \rangle \in \text{traces}(P/s)$  and  $s\hat{\phantom{s}}\langle e \rangle < h$ .

Given the definitions of terminating and nonterminating behaviors, we are now in a position to define another attribute of a process, its behaviors:

**Definition 3.7**  $\text{Behaviors}(P)$  is the set of all  $h$  such that  $h$  is either a terminating behavior of  $P$  or a nonterminating behavior of  $P$ .

We note that, since behaviors are defined entirely in terms of the process, we have not materially extended the mathematics of CSP. However, we now have the means to express eventuality assertions about processes. We note in passing that the set of behaviors of a CSP process can be used to support the interpretation of computation-oriented temporal and interval temporal logics that base their semantics on sequences.

Of the mechanisms for combining CSP processes described in [13], *parallel composition* is the only one that allows processes to interact with one another. So, it is important that the structural relationships between the traces and behaviors of the constituent processes and the traces and behaviors of their parallel composition be well understood. The basic law defining the traces of the composition of two processes in terms of the traces of the components asserts that the result of projecting a trace of the composite onto the alphabet of a constituent results in a trace of the constituent (see [13], p. 72):

$$\text{traces}(P||Q) = \{t \mid t|_{\alpha P} \in \text{traces}(P) \text{ and } t|_{\alpha Q} \in \text{traces}(Q) \text{ and } t \in (\alpha P \cap \alpha Q)\}.$$

It follows easily from this law that the two constituent traces, when projected onto the other processes' alphabet, look the same:

$$\begin{aligned} t|_{\alpha P}|_{\alpha Q} &= t|_{(\alpha P \cap \alpha Q)} \\ &= t|_{(\alpha Q \cap \alpha P)} \\ &= t|_{\alpha Q}|_{\alpha P} \end{aligned}$$

by two applications of law L6 p. 44 of [13]. Two traces that have this property are said to be *parallel compatible*:

**Definition 3.8**  $t_P \in \text{traces}(P)$  and  $t_Q \in \text{traces}(Q)$  are parallel compatible if  $t_P|_{\alpha Q} = t_Q|_{\alpha P}$ .

We extend this definition to behaviors by using their finite initial subsequences:

**Definition 3.9** Any  $h_P \in \text{behaviors}(P)$  and  $h_Q \in \text{behaviors}(Q)$  are parallel compatible if  $\#h_P = \#h_Q$  and for all  $t_P \leq h_P, t_Q \leq h_Q$ :  $\#t_P = \#t_Q \Rightarrow t_P$  and  $t_Q$  are parallel compatible.

One major means of denying service to a process is to prevent its progress. For a process to prevent the progress of another process, it is necessary that there be some kind of mutual interaction in which the process to be blocked depends on some behavior on the part of the blocker in order to continue. Before we define the concept of temporal dependency, however, we must first digress a little to discuss a subtle point of the CSP notation.

The symbols that comprise the alphabet of a CSP process are often referred to as *events* in the CSP literature. However, it is necessary to understand that these symbols in fact are names for event *classes*, of which there may be any number of *occurrences* during the evolution of the process ([13], p. 23). Hence a logic expression like  $x = E$ , where  $x$  denotes an event occurrence and  $E$  denotes a class of event occurrences, may be understood as meaning  $x \in E$ .

As for substitutivity of equals (referential transparency), it fails in general when "pivoting" on an event name:

$$x = E, y = E$$

does not imply  $x = y$ . When transformed, the premises above are

$$x \in \text{foo}, y \in \text{foo}$$

from which nothing interesting can be concluded.

Suppose, however, that the expression  $e$  denotes an event occurrence. Then the deduction

$$x = E, x = e \vdash e = E,$$

proceeds by substitution of equals for equals. When transformed, the resulting rigorous formulation

$$x \in \text{foo}, x = e \vdash e \in \text{foo}$$

does in fact hold by virtue of substitution of equals for equals.

There is a generalization of this situation which needs to be treated. The question is, how is a predicate in one or more event names to be handled? In general, a predication

$$P(E_1, \dots, E_n)$$

can be replaced by the formula

$$\exists e_1 \dots \exists e_n: \bigwedge_{i=1}^n (e_i \in E_i) \wedge P(e_1, \dots, e_n)$$

Given the above it is clear that a trace is simply a (0-based) sequence of event occurrences. Thus an expression like  $y_0 = \text{foo}$  means simply  $y_0 \in \text{foo}$ .

When we select a particular event occurrence, say the  $i^{\text{th}}$   $e_i$ , somewhere along a particular behavior of a process, we see that all the events in the future of  $e_i$  depend on  $e_i$  in the sense that they cannot occur until after  $e_i$  does:

**Definition 3.10** The expression " $e_i \xrightarrow{h} f_j$ " means that  $f_j$ , the  $j^{\text{th}}$  occurrence of event  $f$ , depends on  $e_i$ , the  $i^{\text{th}}$  occurrence of event  $e$ , along a behavior  $h$ .

This basic form of *temporal dependence* is an immediate consequence of the event-driven nature of CSP processes.

It follows easily from the definition that temporal dependence forms a strict partial order relation on event occurrences along a behavior. Temporal dependence is clearly irreflexive since an event cannot depend on itself: For any occurrence  $i$  of event  $e \in \alpha P$  along behavior  $h$ ,  $\neg e_i \xrightarrow{h} e_i$ . Transitivity of temporal dependence follows immediately: For any event occurrences  $e_i$ ,  $f_j$ , and  $g_k$  along a behavior  $h$ , if  $e_i \xrightarrow{h} f_j$  and  $f_j \xrightarrow{h} g_k$  then  $e_i \xrightarrow{h} g_k$ . Finally, the asymmetry of temporal dependence follows because behaviors are not circular: For any event occurrences  $e_i$  and  $f_j$  along  $h$ , if  $e_i \xrightarrow{h} f_j$  then  $\neg f_j \xrightarrow{h} e_i$ .

In CSP the only mechanism that allows two processes to interact is *parallel composition*, in which two processes with intersecting alphabets interleave events not in the intersection and lock-step synchronize single occurrences of events in the intersection. The simplest way to have any dependency between event occurrences  $e_i$  along  $h_P$  in process  $P$  and  $f_j$  along  $h_Q$  in process  $Q$  is for either  $e_i$  or  $f_j$  to be an occurrence of an event in the intersection of the two alphabets  $\alpha P$  and  $\alpha Q$ , but not both:

**Definition 3.11**  $f_j$  along  $h_Q$  in process  $Q$  depends on  $e_i$  along  $h_P$  in process  $P$  if either  $e_i \in \alpha P \cap \alpha Q$  and  $f_j \notin \alpha P$  and  $e_i \xrightarrow{h_Q} f_j$ , or  $f_j \in \alpha P \cap \alpha Q$  and  $e_i \notin \alpha Q$  and  $e_i \xrightarrow{h_P} f_j$ .

A common scenario involving three event occurrences  $e_i$  along behavior  $h_P$  of process  $P$  and  $f_j$  along  $h_Q$  of process  $Q$  is for there to be some event occurrence  $g_k$  in the intersection of their alphabets which acts as an intermediary in their parallel composition:  $e_i \xrightarrow{h_P} g_k$  and  $g_k \xrightarrow{h_Q} f_j$ .

The occurrences of common events in parallel processes supply the “bridges” by which the temporal dependence partial order of component processes are extended into a temporal dependence partial order of the composed system. The fact that the projection of a parallel process’ trace on the alphabet of any component process yields a trace of the component implies that the temporal dependencies of the component carry over into the composite system.

**3.2.1.2 Modes of Service Denial**— We now define a variety of modes or patterns of service denial which arise when some “malicious” processes deny a liveness property to a “victim” process, and which are best described in terms of temporal dependencies between the victim and the malicious processes. We first show how behaviors can be used to express eventuality or liveness properties of a process. We then discuss three common modes of service denial—cyclic deadlock, starvation, and mutual starvation. The three modes of service denial are defined in terms of temporal dependencies among sets of event occurrences, and are defined with respect to a single behavior. Thus it will always be possible for a process which exhibits service denial along one behavior to evolve without incident along other behaviors. It will then be simple to negate the statement of the existence of a particular service denial pattern along a particular behavior to arrive at a policy statement of no service denial along any behavior.

Any point in a particular behavior  $h$  defines a “now” with respect to which  $h$  may be divided into a “past” and a “future”:

**Definition 3.12** For any  $h \in \text{behaviors}(P)$  and any  $t \leq h$ ,  $\text{future}(t, h)$  is the complement in  $h$  of  $t$ .

$\text{future}(t, h)$  is well-defined, although it may be the empty sequence if  $t = h$ . When  $h$  is a nonterminating behavior and  $t < h$  then  $\text{future}(t, h)$  is infinite.

Given the notion of behaviors of a process, we can now assert the guaranteed occurrence of an event  $e$  sometime in the lifetime of process  $P$ :



$\forall h \in \text{behaviors}(P) : \langle e \rangle \text{ in } h,$

where “ $e$  in  $h$ ” means  $e$  is a contiguous subsequence of  $h$ ; in other words, for some  $s, t \in A^\omega : h = s \hat{~} e \hat{~} t$ . Notice that in this expression the number of events before the first occurrence of  $e$  may differ for each  $h$ ; if there are an infinite number of different behaviors then there may be no upper bound on the “distance” to the first occurrence of  $e$ .

A weaker guarantee of event occurrence is useful for expressing liveness properties of systems in which the guarantee of eventual occurrence of an event  $f$  is contingent on the occurrence of another event  $e$ : For all  $h \in \text{behaviors}(P), t < h$ : if  $\bar{t}_0 = e$  then  $\langle f \rangle \text{ in future}(t, h)$  [18]

Now that we can speak of the future of a trace it possible to express the idea of an event occurring *infinitely often*. The proposition that  $e$  happens infinitely often on all behaviors of process  $P$  is captured in the temporal logic expression “ $\Box \Diamond e$ ” (see [16], p. 240)<sup>4</sup> This can be translated into CSP by noting that  $\Box \Diamond e$  means that  $\Diamond e$  holds for any point in the future; in other words, given any point of time, there is a later occurrence of  $e$ . So, the notion of occurring infinitely often can be captured in CSP as:

**Definition 3.13**  $e$  occurs infinitely often in behavior  $h$  of process  $P$  if and only if

$$h \notin A^* \wedge \forall t : t < h \Rightarrow \langle e \rangle \text{ in future}(t, h),$$

From this definition, the following theorem is obvious:

**Theorem 1**  $e$  occurs infinitely often in behavior  $h$  of process  $P$  if and only if the sequence  $h$  contains infinitely many occurrences of  $e$ .

Thus, one of the advantages of using CSP is that the notion of an event occurring infinitely often has a much simpler definition than it does in other formalisms.

Karp, among others, uses the concept of an event occurring infinitely often to define *fairness* [16].

**Definition 3.14** Let  $S$  be a process, let  $e, f$  be members of  $\alpha S$ , and suppose that an occurrence of  $e$  is a necessary but not sufficient condition for a subsequent occurrence of  $f$ .  $S$  is fair for  $f$  with respect to  $e$  if for all  $h \in \text{behaviors}(S)$ , if  $e$  occurs infinitely often in  $h$  then  $\langle f \rangle \text{ in } h$ .

The name *fairness* comes from the field of operating systems, where  $e$  is the event that some process becomes *ready*, and  $f$  is the event that the process is dispatched; this formalizes the statement, if a process is ready often enough, it will eventually be dispatched. Notice that the negation of fairness defines *livelock*: for some  $h \in \text{behaviors}(P)$ ,  $e$  occurs infinitely often in  $h$  but  $\neg(\langle f \rangle \text{ in } h)$ .

It can happen that a pair of event occurrences has opposite orders of temporal dependency in two different processes:

---

<sup>4</sup>See Section 2.4 for a description of  $\Box$  and  $\Diamond$ .

**Definition 3.15** Let  $P$  and  $Q$  be CSP processes with behaviors  $h_P$  and  $h_Q$  respectively, and let events  $e, f \in \alpha P \cap \alpha Q$ : Event occurrences  $e_i$  and  $f_j$  are cyclically dependent if both  $e_i \xrightarrow{h_P} f_j$  and  $f_j \xrightarrow{h_Q} e_i$ .

When we form the parallel composition  $(P||Q)$  of  $P$  and  $Q$  above and consider the behaviors of  $(P||Q)$  that arise from  $h_P$  and  $h_Q$  (and we presume that some exist), we find that there exist sequences  $t_P \in \alpha P^*$ ,  $x \in \alpha P^*$ ,  $s_P \in \alpha P^\omega$ ,  $t_Q \in \alpha Q^*$ ,  $y \in \alpha Q^*$ , and  $s_Q \in \alpha Q^\omega$  such that

$$h_P = t_P \cdot \langle e_i \rangle \cdot x \cdot \langle f_j \rangle \cdot s_P,$$

$$h_Q = t_Q \cdot \langle f_j \rangle \cdot y \cdot \langle e_i \rangle \cdot s_Q,$$

and the behaviors of  $(P||Q)$  that arise from  $h_P$  and  $h_Q$  are the members of the set:

$$\{t : t|_{\alpha P} = t_P \text{ and } t|_{\alpha Q} = t_Q\}.$$

Since no trace is longer than  $\#t_P + \#t_Q$ ,  $(P||Q)$  terminates. Thus the composed process is blocked with  $P$  waiting to participate in  $e_i$  and  $Q$  waiting to participate in  $f_j$ . This situation is called *cyclic deadlock*. We must keep in mind the fact that we are considering those behaviors of  $(P||Q)$  that arise when  $P$  attempts to progress along  $h_P$  and  $Q$  attempts to progress along  $h_Q$ ; there may well be other evolutions of  $P$  and  $Q$  for which  $(P||Q)$  will evolve forever.

For example, suppose process  $P$  works by participating in some local event  $a$ , and then sending a message on channel  $c$  followed by awaiting an answer on channel  $d$ :

$$\alpha P = \{a\} \cup \alpha c \cup \alpha d$$

$$P = \mu X.(a \rightarrow c!x \rightarrow d?y \rightarrow X),$$

and suppose process  $Q$  is supposed to work by participating in event  $b$  followed by receiving a message on channel  $c$  and sending a reply on channel  $d$ . However, a design error results in inverting the order of the communication protocol events:

$$\alpha Q = \{b\} \cup \alpha c \cup \alpha d$$

$$Q = \mu X.(b \rightarrow d!y \rightarrow c?x \rightarrow X).$$

When these two processes are composed in parallel, cyclic deadlock occurs on any behavior. The relevant event occurrences are  $c.x_1$  and  $d.y_1$  (recall that in CSP the forms " $x!y$ " and " $x?y$ " are syntactic sugar for the communication event " $x.y$ ", designed to indicate the direction of information flow; see [13], chapter 4). In any behavior  $h_P$  of  $P$   $c.x_1 \xrightarrow{h_P} d.y_1$ , while in any behavior  $h_Q$  of  $Q$   $d.y_1 \xrightarrow{h_Q} c.x_1$ . Thus  $P||Q$  hangs with  $P$  awaiting  $c.x_1$  and  $Q$  awaiting  $d.y_1$ .

A cyclic deadlock situation can be composed of any number of event occurrences forming a dependency cycle. For example, if  $e \in \alpha P \setminus \alpha Q$ ,  $f \in \alpha Q \setminus \alpha P$ ,  $x, y \in \alpha P \cap \alpha Q$ , and  $x_i \xrightarrow{h_P} e_j$ ,  $e_j \xrightarrow{h_P} y_k$  in a behavior  $h_P$  of  $P$  and  $y_k \xrightarrow{h_Q} f_l$ ,  $f_l \xrightarrow{h_Q} x_i$  in a behavior  $h_Q$  of  $Q$ , then there is a cyclic dependency, so  $P||Q$  deadlocks with  $P$  waiting to participate in  $x_i$  and  $Q$  waiting to participate in  $y_k$ . In this case it is easy to see the dependency, since deleting the two "local" event occurrences  $e_i$  and  $f_l$  leaves the basic situation of two common events that must occur in opposite order in the two processes.

When more processes are introduced, cyclic dependency may no longer be evident between any two processes. For example, consider three processes  $P$ ,  $Q$ , and  $R$  with events  $e \in \alpha P \cap \alpha Q$ ,  $f \in \alpha Q \cap \alpha R$ , and  $g \in \alpha R \cap \alpha P$ : If  $e_i \xrightarrow{h_P} g_k$  in a behavior  $h_P$  of  $P$ ,  $f_j \xrightarrow{h_Q} e_i$  in a behavior  $h_Q$  of  $Q$ , and  $g_k \xrightarrow{h_R} f_j$  in a behavior  $h_R$  of  $R$ , then there is a cyclic dependency and  $P \parallel Q \parallel R$  will deadlock with  $P$ ,  $Q$ , and  $R$  waiting to participate in  $e_i$ ,  $f_j$ , and  $g_k$  respectively. However, examination of any pair will only reveal a single dependency, not a cycle; thus any two processes could proceed, but the three processes are cyclically deadlocked.

All the members of a set of processes that are in cyclic deadlock are prevented from making further progress. Starvation, on the other hand, is a form of service denial in which some processes make progress themselves while preventing some "victim" process from progressing. Let  $P$  and  $Q$  be processes, let  $t_Q$  be a trace and  $h_Q$  be a behavior of  $Q$  such that  $t_Q \cdot e_i \leq h_Q$  for some  $e_i \in \alpha Q$ , and let  $t$  be a trace of  $P$  that is *parallel compatible* with  $t_Q$ , i.e. so that  $t_Q \mid \alpha P = t \mid \alpha Q$ : If  $h_{P/t}$  is a behavior of  $P/t$  such that  $h_{P/t} \mid \alpha Q = \langle \rangle$  then  $Q$  is starved by  $P$  along any behavior  $h$  of  $P \parallel Q$  composed from  $t_P \cdot h_{P/t}$  and  $t_Q$ . Roughly speaking,  $P \parallel Q$  proceeds along until the point at which  $Q$  needs to participate in  $e_i$ , at which time  $P$  proceeds along a behavior which nevermore interacts with  $Q$ .

For example, suppose process  $P$  is passing a buffer back and forth with process  $Q$  and then at some point  $P$  refuses to return it and proceeds along its own private way, so  $Q$  starves forever:

$$\begin{aligned} \text{COOPERATE}_n &= p.\text{stuff} \rightarrow \text{pass.buffer} \rightarrow \text{COOPERATE}_{n-1}, n > 0, \\ \text{COOPERATE}_0 &= \text{NO.FURTHER.INTERACTION}, \\ P &= \text{COOPERATE}_k, \text{ and} \\ Q &= \mu X. (q.\text{stuff} \rightarrow \text{pass.buffer} \rightarrow X) \end{aligned}$$

After  $k$  cycles of cooperation  $P$  does not participate in  $\text{pass.buffer}_{k+1}$ , and  $Q$  waits forever for participation in  $\text{pass.buffer}_{k+1}$ . In the definition given above,  $t_Q$  is the trace of  $Q$  up to  $\text{pass.buffer}_{k+1}$  and  $t$  is the trace of  $P$  up through  $\text{pass.buffer}_{k+1}$ .

A hybrid situation can arise when process  $P$  and process  $Q$  both try to participate in events from their common alphabet but the events are different. As a result, the parallel system is deadlocked.

To see how this differs from cyclic dependence, suppose  $P$  and  $Q$  are defined as follows:

$$\begin{aligned} P &= e \rightarrow \text{STOP}_{\{e,f\}} \\ Q &= g \rightarrow f \rightarrow \text{STOP}_{\{e,f,g\}} \end{aligned}$$

Initially  $P$  is trying to participate in event  $e$ . Since  $e \in \alpha Q$ , it cannot do so until  $Q$  is ready to participate in  $e$ , too. But  $Q$  is never ready to participate in  $e$ , so  $P$  is starved by  $Q$ . After participating in  $g$ ,  $Q$  tries to participate in  $f$ . Because  $P$  never proceeds, it cannot participate in  $f$ , so  $Q$  becomes starved by  $P$ . Yet, there is no cyclic dependency since  $g \xrightarrow{(g,f)} f$  is the only dependency.

So, cyclic deadlock is different from mutual starvation, although it is clear that any instance of cyclic deadlock is an instance of mutual starvation.

Our definition of mutual starvation is consistent with the definition of deadlock given in [13]. This means that cyclic deadlock is a proper subclass of the type of deadlock defined in [13] and that we can distinguish between cyclic deadlock and other types of deadlock while the definition in [13] does not allow such a distinction to be made.

Starvation is not a subclass of mutual starvation since one of the processes continues execution in the case of starvation while both processes are blocked in the case of mutual starvation. It is also clear that mutual starvation is not a subclass of starvation; consequently, the concepts are mutually exclusive. This means that starvation is not addressed by the definition of deadlock given in [13]. Although the system can still perform useful processing when starvation occurs, there are instances in which it is undesirable for certain processes to be starved. Thus, it is important to have a formal definition of starvation that allows occurrences of it to be identified.

**3.2.1.3 Service Denial Policies** — The notions of dependency, deadlock, and starvation have been defined with respect to particular occurrences of events. To state predicates concerning dependency relations we need to extend these definitions to repeatable or universal properties expressed in terms of events (strictly, event classes, as opposed to event occurrences). We first extend the idea of temporal dependency to events in a particular behavior. In the following, let  $P = \langle A, F, D \rangle$  be a process,  $h$  be a behavior of  $P$ , and  $e$  and  $f$  be events in  $A$ .

We say that event  $f$  depends on event  $e$  in a behavior  $h$  if each new occurrence  $f_i$  of  $f$  is preceded by a new occurrence  $e_i$  of  $e$ :

**Definition 3.16**  $f$  (temporally) depends on  $e$  in  $h$  if and only if

$$\forall t \leq h: t \in A^* \Rightarrow t \downarrow f \leq t \downarrow e$$

We note that this definition allows there to be arbitrarily many occurrences of the event depended on before the next occurrence of the dependent event. It is possible to restrict the degree of "buffering ahead" that is possible, but we will not explore this further.

This behavior-specific relationship can be strengthened to dependency over all behaviors:

**Definition 3.17**  $f$  (temporally) depends on  $e$  in  $P$  if and only if

$$\forall h \in \text{behaviors}(P), \forall t \leq h: t \in A^* \Rightarrow t \downarrow f \leq t \downarrow e$$

We have the result

**Theorem 2** If  $f$  depends on  $e$  in  $h$  then  $\forall i > 0: e_i \xrightarrow{h} f_i$

and similarly over all behaviors.

As discussed in [13], pp. 161–170, a convenient way to model the sharing of a resource by client processes is by interleaving the client processes, composing the interleaved clients in parallel with

the resource, and hiding the alphabet of the resource. The result is a compound process in which the resource access is hidden from the externally visible event stream. The use of interleaving allows the protocol for accessing the resource to be made explicit, so that properties of coherent resource access may be formulated and (hopefully) proven of a system model. If it is desired to have the clients access a set of resources, then all the resources are composed in parallel (usually with disjoint alphabets), and the resulting composite replaces "the resource" in the description above.

Since the interleaved clients cannot interact directly with one another, the only way to obtain any sort of temporal dependence among the clients is for an event occurrence  $d_i$  along  $h_P$  in client process  $P$  to depend on an event occurrence  $g_l$  along  $h_Q$  in client process  $Q$  by having intermediary event occurrences  $e_j$  and  $f_k$  in the resource process  $R$  such that  $d_i \xrightarrow{h_P \parallel R} e_j$ ,  $e_j \xrightarrow{h_R} f_k$ , and  $f_k \xrightarrow{h_Q \parallel R} g_l$ . One way to obtain this relationship is described in

**Theorem 3** *Let  $P$  and  $Q$  be processes with  $\alpha P = \alpha Q = A$ ,  $R$  be a resource process with  $\alpha R \subseteq A$ ,  $d$  and  $g$  be events  $\in A \setminus \alpha R$ , and  $e$  and  $f$  be events  $\in \alpha R$ .*

*If  $e$  depends on  $d$  in  $P$ ,  $f$  depends on  $e$  in  $R$ , and  $g$  depends on  $f$  in  $Q$ , then  $g$  depends on  $d$  in  $R/(P \parallel Q)$*

Since a common source of deadlock in real-systems is the result of shared resources, it is important to have a theorem such as this that allows the detection of potential deadlocks through the sharing of resources.

We can identify several levels of "sensitivity" of liveness contingent on another event (see 3.2.1.2), as discussed in [8]. The strongest form obtains when an event  $f$  is guaranteed to occur merely on the single occurrence of event  $e$ . When this happens, we have that the future of the  $i^{th}$  occurrence of  $e$  contains the  $i^{th}$  occurrence of  $f$ :

$$\forall h, t, i : t < h \wedge t \downarrow e = i \wedge \bar{t}_0 = e \Rightarrow \text{future}(t, h) \downarrow f \geq 1 \wedge t \downarrow f = i - 1.$$

This strong guarantee might be appropriate for responsiveness to internally generated events, where some control can be designed in. However, it is likely to be too strong for externally generated events which may arrive at awkward times when immediate response is impossible.

A somewhat weaker form of the liveness guarantee merely guarantees that if the triggering event occurs *infinitely often* the response event will eventually occur. This may best be understood negatively as saying that there is no behavior along which the triggering event occurs infinitely often while the desired event never occurs:

$$\forall h, t, v : v = \text{future}(t, h) \Rightarrow ((\forall s : s < v \Rightarrow e \text{ in } \text{future}(s, v)) \Rightarrow f \text{ in } v)$$

To understand what this means, suppose  $t$  is a trace of the system and let  $v$  be an arbitrary sequence so that  $t \hat{\ } v$  is a behavior of the system. Then, the requirement is that whenever  $e$  occurs infinitely often in  $v$ , then  $f$  occurs in  $v$ . So, no matter what point is reached in the processing, it is always the case that there cannot subsequently be the occurrence of an infinite number of  $e$ 's without there being at least one occurrence of  $f$ .

This guarantee of responsiveness would be appropriate for example for a process waiting on a semaphore.

Along with a statement of liveness it may be desirable to require that there be no spurious responses, i.e. responses that were not "caused" by the triggering event (we place "cause" in quotes because there is no full-blown concept of causality here, and we do not define one). This restriction is well expressed by the extension of temporal dependency to event classes that was introduced at the beginning of section 3.2.1.3. The statement that  $f$  temporally depends on  $e$  in behavior  $h$  captures the idea that the  $i^{th}$  occurrence of  $f$  is guaranteed to be preceded by the  $i^{th}$  occurrence of  $e$  in  $h$ .

The most basic service assurance policy with respect to cyclic deadlock would be the assertion that no such pattern of event occurrences exists on any behavior of the system:

### Cyclic Deadlock Policy

$$\forall h_P, h_Q, e, f, i, j : e_i \xrightarrow{h_P} f_j \Rightarrow e_i \xrightarrow{h_Q} f_j.$$

While effective, this may not be feasible. In particular, it does not take into account that the behavior of user processes may not be controllable. To address this, we must weaken the policy so that rather than requiring that there be no deadlock, it requires that only certain processes be deadlocked.

### Conditional Cyclic Deadlock Policy

$$\forall h_P, h_Q, e, f, i, j : e_i \xrightarrow{h_P} f_j \Rightarrow e_i \xrightarrow{h_Q} f_j.$$

unless  $P$  and  $Q$  are permitted to be deadlocked.

This assumes that a relation has been defined on the processes that indicates which processes may be deadlocked. For example, it might be reasonable to require that user processes and system services can never become deadlocked. Then, the above policy would prevent system services from becoming deadlocked with user processes while ignoring the possibility of user processes becoming deadlocked with each other.

Basic service assurance policies for starvation and mutual starvation can be derived in a similar fashion to that for cyclic deadlock.

### Starvation Policy

$$\forall h_P, h_Q, e, i, t_P : t_P \hat{e}_i < h_P \Rightarrow \exists t_Q : t_Q \hat{e}_i < h_Q.$$

### Mutual Starvation Policy

$$\forall h_P, h_Q : h_P | \alpha Q = h_Q | \alpha P$$

Once again these policies can be conditionalized to be less stringent.

### Conditional Starvation Policy

$$\forall h_P, h_Q, e, i, t_P : t_P \hat{e}_i < h_P \Rightarrow \exists t_Q : t_Q \hat{e}_i < h_Q.$$

unless  $P$  is permitted to starve  $Q$ .

### Conditional Mutual Starvation Policy

$$\forall h_P, h_Q : h_P | \alpha Q = h_Q | \alpha P$$

unless  $P$  and  $Q$  are permitted to starve each other.

As with the cyclic deadlock policy, a relation must be defined specifying when a process can starve another process. One possibility would be to allow system services from starving user processes while prohibiting user processes from starving system services.

Although the discussion in this section is given in terms of the model for nondeterministic processes in CSP, the policies developed are only applicable for deterministic processes in CSP since they make little use if any of process attributes other than traces. Further research needs to be done to determine the degree to which real-systems are nondeterministic in the CSP sense. If it is determined that it is common for real-systems to be nondeterministic in the CSP sense, that the definitions in this section must be generalized to address nondeterministic processes. This will require adequately addressing the failures ( $F$ ) and divergences ( $D$ ).

If  $D$  is nonempty, then it is possible for the process to enter states from which it is impossible to know what processing will subsequently be done. Essentially, there is a trace  $s$  such that the behavior of the process after  $s$  is undefined<sup>5</sup>. Clearly it is not possible to show that deadlock does not occur after  $s$  since there is no information available concerning the behavior after  $s$ . Consequently,  $D$  can be addressed by requiring that it be empty.

Now, consider  $F$ . Suppose  $P$  is a process such that:

- $(s_P, X) \in F_P$
- $e \in X$
- $(s_P \hat{e}, \{\}) \in F_P$ <sup>6</sup>
- $(s_P \hat{f}, \{\}) \in F_P$

and suppose  $P$  is composed with a process  $Q$  such that:

- $(s_Q, \{\}) \in F_Q$

<sup>5</sup>Although technically the behavior is defined, it is defined to be unconstrained. Thus, no useful assertions can be made about the future behavior

<sup>6</sup>Note that  $(t, \{\}) \in F$  if and only if  $t$  is a trace.

- $s_Q | \alpha P = s_P | \alpha Q$
- $(s_Q^{\sim}(e), \{\}) \in F_Q$

After  $P$  and  $Q$  participate in  $s_P$  and  $s_Q$ , respectively,  $Q$  can only participate in  $e$  if  $P$  is also willing to participate in  $e$ . Since  $e \in X$ , where  $(s_P, X) \in F_P$ ,  $P$  can choose to refuse to participate in  $e$  even though  $Q$  is attempting to participate in  $e$ . For example,  $s_P^{\sim}(f)$  is a trace for  $P$ , so  $P$  could participate in  $f$ . This would result in  $Q$  being blocked. If  $P$  depends on interaction with  $Q$  subsequent to  $s_P$ , then mutual starvation would occur. This is addressed by our definition since  $s_P^{\sim}(f)$  is not parallel compatible with  $s_Q^{\sim}(e)$ . If  $P$  does not require further interaction with  $Q$ , then only  $Q$  is starved. This is addressed by our definition since  $P$  can progress after participating in  $f$  even though  $Q$  is blocked.

Now, suppose  $P$  and  $Q$  are as before except we assume  $e$  is not in any  $X$  for which  $(s, X) \in F_P$ . Then,  $P$  cannot refuse  $e$  when  $Q$  attempts to participate in  $e$ . So, even though our definitions identify the potential mutual starvation or starvation described above, the mutual starvation and starvation are avoided since  $P$  is "intelligent" enough to avoid behaviors that are not consistent with  $Q$ . Because our definitions do not make full use of the set  $F$ , they do not distinguish between this case in which there is no mutual starvation or starvation and the previous case in which there either is mutual starvation or starvation. To distinguish between these cases, define:

**Definition 3.18** A behavior  $h_P$  of  $P$  is inconsistent with a behavior  $h_Q$  of  $Q$  if and only if:

Given  $t_P, t_Q, e, f$  such that:

- $t_P^{\sim}(e) < h_P$
- $t_Q^{\sim}(f) < h_Q$
- $t_P$  is parallel compatible with  $t_Q$  (i.e.  $t_P | \alpha Q = t_Q | \alpha P$ )
- $e, f \in (\alpha P) \cap (\alpha Q)$
- $e \neq f$

then either:

- $(t_P^{\sim}(f), \{\}) \in F_P$  and  $(t_P, f) \notin F_P$ , or
- $(t_Q^{\sim}(e), \{\}) \in F_Q$  and  $(t_Q, e) \notin F_Q$

In other words, there is some point at which  $h_P$  and  $h_Q$  are in disagreement, even though the processes are prohibited from disagreeing at that point.

The parallel composition of processes is defined so that inconsistent behaviors are prevented from occurring simultaneously. Thus, there is no concern in the case in which  $P$  and  $Q$  experience starvation or mutual starvation through  $h_P$  and  $h_Q$  with respect to our earlier definitions unless  $h_P$  and  $h_Q$  are consistent.



Based on the above discussion, it seems reasonable to change our earlier definitions so that they require the pairs of behaviors of  $P$  and  $Q$  considered to be consistent. Further research is required to determine if this is adequate to address systems that are nondeterministic in the CSP sense.

### 3.2.2 Example Real-Time Policy

In this section we provide a worked example of a real-time policy. Our example is based on the elevator example provided in *Algorithms for Fault Tolerant Distributed Systems* [17]. In [17], the elevator and its real-time policy are specified in a flavor of interval temporal logic (ITL).

**3.2.2.1 Elevator Model**— In this section, the elevator system is specified in terms of time-dependent state predicates and ITL. The ITL specifications in this section are copied from [17]. There are some errors in the model in [17] that have been corrected here and show up as deviations between the ITL specification and the other specifications. We discuss these deviations as they are encountered. The next section defines state predicates that we use in the specification of the elevator. The subsequent sections provide the model, policy, and analysis for the elevator.

We are not developing an RTL model because it is very difficult to phrase some of the assertions in RTL. For example, consider the assertion that the elevator is never on more than one floor at a time. In RTL, we can represent this by defining state predicates that test whether the elevator is on each floor. Then, we can say that the elevator is on floor  $i$  at a given time if it reached floor  $i$  at an earlier time but did not leave floor  $i$  until a later time. Now consider the following assertions:

- The elevator is on floor  $i$  at a time if the elevator previously reached the floor for the  $j^{th}$  time and subsequently left the floor for the  $j^{th}$  time.
- The elevator is on floor  $i$  at a time if the elevator previously reached the floor for the  $j^{th}$  time and subsequently left the floor for the  $j + 1^{st}$  time.

Until we know whether the elevator was initially on floor  $i$ , there is no way to determine which assertion is correct. We do not feel it is worthwhile adding implementation detail to the model simply to provide an RTL example of the elevator.

But, we do feel that the notion of state predicates in RTL is worthwhile. So, we develop a state predicate model. The state predicate model is a very primitive model in which we can use state predicates to determine the state of the system at any given time. We use  $S(t)$  to test the value of state predicate  $S$  at time  $t$ .

We use the following primitive state predicates:

- $above\_floor(i, t)$  — indicates whether the elevator is above floor  $i$  and below floor  $i + 1$  at time  $t$ ,
- $doors\_closed(i, t)$  — indicates whether the doors on floor  $i$  are closed at time  $t$ ,

- *doors\_closing*(*i*, *t*) — indicates whether the doors on floor *i* are closing but not yet closed at time *t*,
- *doors\_obstructed*(*i*, *t*) — indicates whether the doors on floor *i* are obstructed at time *t*,
- *doors\_open*(*i*, *t*) — indicates whether the doors on floor *i* are open at time *t*,
- *doors\_opening*(*i*, *t*) — indicates whether the doors on floor *i* are opening but not yet open at time *t*,
- *down\_requested*(*i*, *t*) — indicates whether the down button on floor *i* is pressed at time *t*,
- *floor\_requested*(*i*, *t*) — indicates whether the button corresponding to floor *i* in the elevator is pressed at time *t*,
- *going\_up*(*t*) — indicates whether the elevator is either travelling upwards or intending to travel upwards at time *t*,
- *lift\_closed*(*t*) — indicates whether the lift doors are closed at time *t*,
- *lift\_closing*(*t*) — indicates whether the lift doors are closing but not yet closed at time *t*,
- *lift\_obstructed*(*t*) — indicates whether the lift doors are obstructed at time *t*,
- *lift\_open*(*t*) — indicates whether the lift doors are open at time *t*,
- *lift\_opening*(*t*) — indicates whether the lift doors are opening but not yet open at time *t*,
- *on\_floor*(*i*, *t*) — indicates whether the elevator is on floor *i* at time *t*,
- *up\_requested*(*i*, *t*) — indicates whether the up button on floor *i* is pressed at time *t*,

We also use the following derived state predicates<sup>7</sup>:

- *down\_light*(*i*, *t*) — indicates whether the down button on floor *i* has been pressed without the elevator subsequently servicing floor *i*; more formally, this predicate evaluates to true if and only if:

$\exists t_1$  such that :

*down\_requested*(*i*, *t*<sub>1</sub>)

and *doors\_closed*(*i*, *t*<sub>1</sub>)

and  $\forall t_2 \in (t_1, t] : \neg \textit{servicing}(i, t_2)$

- *floor\_light*(*i*, *t*) — indicates whether the button corresponding to floor *i* in the elevator has been pressed without the elevator subsequently servicing floor *i*; more formally, this predicate evaluates to true if and only if:

$\exists t_1$  such that :

*floor\_requested*(*i*, *t*<sub>1</sub>)

---

<sup>7</sup>The derived state predicates are defined in terms of the primitive state predicates.

and *doors\_closed*(*i*, *t*<sub>1</sub>)

and  $\forall t_2 \in (t_1, t] : \neg \textit{servicing}(i, t_2)$

- *request\_light*(*i*, *t*) — indicates whether a request is pending for floor *i* at time *t*; more formally, this predicate evaluates to true if and only if:

*up\_light*(*i*, *t*)

or *down\_light*(*i*, *t*)

or *floor\_light*(*i*, *t*)

- *reverse*(*t*) — indicates that the elevator has changed the direction it intends to travel at time *t*; more formally, this predicate evaluates to true if and only if:

$\exists v, t_1$  such that  $t_1 < t$  :

$\forall t_2 \in (t_1, t) : (\textit{going\_up}(t_2) \neq \textit{going\_up}(t))$

- *servicing*(*i*, *t*) — indicates whether the elevator is servicing floor *i* at time *t*; more formally, this predicate evaluates to true if and only if:

*on\_floor*(*i*, *t*)

and *lift\_open*(*t*)

and *doors\_open*(*i*, *t*)

- *up\_light*(*i*, *t*) — indicates whether the up button on floor *i* has been pressed without the elevator subsequently servicing floor *i*; more formally, this predicate evaluates to true if and only if:

$\exists t_1$  such that :

*up\_requested*(*i*, *t*<sub>1</sub>)

and *doors\_closed*(*i*, *t*<sub>1</sub>)

and  $\forall t_2 \in (t_1, t] : \neg \textit{servicing}(i, t_2)$

The following constraints on the state predicates describe the desired operation of the elevator. For ease of comparison, we have included both the ITL specification<sup>8</sup> from [17] and our state predicate versions of each constraint. Whenever the ITL version and state predicate version of a constraint are not in agreement, we discuss the reason for the difference. Note that since  $\Rightarrow$  is an interval operator in ITL,  $\supset$  is used to denote logical implication in ITL. To avoid confusion, we use *implies* to denote logical implication in the state predicate specifications.

- Constraints on *on\_floor*:

---

<sup>8</sup>See Section 2.5 for a description of ITL

a. The elevator can never go below floor 0 or above floor  $MAX_{floor}$ .

- ITL:  $\neg atfloor(-1) \wedge \neg atfloor(n+1)$

In addition to requiring that the elevator is never at floor  $-1$  or floor  $n+1$ , it is necessary to require that the elevator not be at other invalid floors. Thus, the model we have developed here has a stronger assertion.

- State Predicates

$\forall i, t :$

$i < 0$

or  $i > MAX_{floor}$

implies

$\neg on\_floor(i, t)$

b. The elevator cannot be on more than one floor at the same time.

- ITL

$b \neq a \wedge atfloor(a) \supset \neg atfloor(b)$

- State Predicates

$\forall i, j, t :$

$on\_floor(i, t)$

and  $i \neq j$

implies

$\neg on\_floor(j, t)$

• Constraints on *above\_floor*:

The constraints relevant to *above\_floor* have no analogues in the ITL specification. In the ITL specification, it is assumed that an elevator is always on some floor. When the elevator moves from one floor to another, it does so instantaneously. This makes it impossible to distinguish between the case in which a button is pressed while the elevator is stationary at a floor and the case in which a button is pressed while the elevator is in motion leaving a floor. As a side-effect of this, it appears that if the elevator is at a floor with its doors closed, a button is pressed at that floor, and no other buttons are ever pressed, that the elevator can simply sit at the floor with its doors closed forever. Since we do not want our elevator to behave like this, it is necessary to allow for the possibility that the elevator is sometimes between floors rather than actually on a floor.

a. The elevator cannot be between one pair of consecutive floors at the same time as it is between another pair of consecutive floors.

$\forall i, j, t :$

$above\_floor(i, t)$

and  $i \neq j$

*implies*

$\neg \text{above\_floor}(j, t)$

b. The elevator can never go below floor 0 or above floor  $MAX_{Floor}$ .

$\forall i, t :$

$i < 0$

or  $i \geq MAX_{floor}$

*implies*

$\neg \text{above\_floor}(i, t)$

- Constraints on *up\_requested*:

a. There is no up button on floor  $MAX_{floor}$  or nonexistent floors.

- ITL

$\neg \text{light}(n, up)$

In addition to requiring that an up request is never made at the top floor, it is necessary to require that an up request is never made for invalid floors. Thus, the model we have developed here has a stronger assertion.

- State Predicate

$\forall i, t :$

$i < 0$

or  $i \geq MAX_{floor}$

*implies*

$\neg \text{up\_requested}(i, t)$

- Constraints on *down\_requested*:

a. There is no down button on floor 0 or nonexistent floors.

- ITL

$\neg \text{light}(0, down)$

In addition to requiring that a down request is never made at the bottom floor, it is necessary to require that a down request is never made for invalid floors. Thus, the model we have developed here has a stronger assertion.

- State Predicate

$\forall i, t :$

$i < 0$

or  $i \geq MAX_{floor}$

*implies*

$\neg down\_requested(i, t)$

- Constraints on *floor\_requested*:

None of the constraints from the ITL specification are relevant, so only the state predicate versions are listed.

- a. There is only a button corresponding to floors 0 through  $MAX_{floor}$  inclusive in the elevator.

$\forall i, t :$

$i < 0$

or  $i > MAX_{floor}$

*implies*

$\neg floor\_requested(i, t)$

- Constraints on elevator movement:

- a. The elevator can move only to adjacent floors.

- ITL

$[atfloor(a) \Rightarrow before (atfloor(a+1) \vee atfloor(a-1))] \Box atfloor(a)$

Actually, this says that if the elevator is at floor  $a$  and is later at either floor  $a - 1$  or floor  $a + 1$ , then during the time between when it is on floor  $a$  and later reaches an adjacent floor, it is always on floor  $a$ . So, if the elevator is at floor 0, this assertion does not prevent it from jumping to floor 2 as long as the elevator never visits floor 1. Although the specification we use in our state predicate model is more complex, it accurately captures the assertion.

- State Predicate

$\forall i, j, t_1, t_2$  such that  $t_1 < t_2 :$

$on\_floor(i, t_1)$

and  $on\_floor(j, t_2)$

and  $\forall k, t \in (t_1, t_2) : \neg on\_floor(k, t)$

*implies*

$j = i + 1$  and  $\forall t \in (t_1, t_2) : above\_floor(i, t)$

or  $j + 1 = i$  and  $\forall t \in (t_1, t_2) : above\_floor(j, t)$

- b. If the elevator is at a floor, a request is pending for some other floor, and the doors are closed, then the elevator begins movement to an adjacent floor.

– ITL

$$b \neq a \supset [(\text{closed}(a) \wedge \text{light}(b, \text{dir})) \Rightarrow (\text{closed}(a) \wedge \text{light}(b, \text{dir})) + \text{movement\_time}] \\ \square \text{in\_service} \supset (* \text{at\_floor}(a+1) \vee * \text{at\_floor}(a-1))^9$$

Note that this ITL assertion also addresses the next requirement. Also, note that the ITL specification allows for the possibility that the elevator might go out of service at any time. We have not addressed this possibility in our model. This is not a serious deficiency in our model. It simplifies the model and really subtracts nothing since the policy developed for the elevator is only relevant to operational elevators. If we instead were interested in a policy such as:

*When the elevator becomes operational after being out of service, it always returns to floor 0.*

then we clearly would need to model the possibility that the elevator is out of service. In any case, it would be straightforward to add the out of service possibility to our model.

– State Predicate

$$\forall t_1, t_2, i, j :$$

$$j \neq i$$

and  $\text{request\_light}(j, t_2)$

and  $\forall t \in (t_1, t_2) : \text{on\_floor}(i, t)$

and  $\text{doors\_closed}(i, t_2)$

*implies*

Either:

$(\text{going\_up}(t_2) \text{ and } \text{above\_floor}(i, t_2))$

or  $(\neg \text{going\_up}(t_2) \text{ and } \text{above\_floor}(i-1, t_2))$

- c. Anytime the elevator is in motion, it will reach the next floor within time *movement\_time* unless it reverses.

– ITL

$$b \neq a \supset [(\text{closed}(a) \wedge \text{light}(b, \text{dir})) \Rightarrow (\text{closed}(a) \wedge \text{light}(b, \text{dir})) + \text{movement\_time}] \\ \square \text{in\_service} \supset (* \text{at\_floor}(a+1) \vee * \text{at\_floor}(a-1))$$

Note that this ITL assertion also addresses the previous requirement.

<sup>9</sup>[17] does not define the meaning of  $*e$  when  $e$  is an event rather than an interval. We suspect  $*e$  is meant to be interpreted as  $\Diamond e$ .

– State Predicate

$\forall t, i :$

$above\_floor(i, t)$

and either:

$(going\_up(t) \text{ and } (\forall t_1 \in (t, t + movement\_time] : \neg on\_floor(i + 1, t_1)))$

or  $(\neg going\_up(t) \text{ and } (\forall t_2 \in (t, t + movement\_time] : \neg on\_floor(i, t_2)))$

*implies*

$\exists t_3 \in (t, t + movement\_time] : reverse(t_3)$

d. The elevator must always be on a floor or between two consecutive floors.

Since the ITL specification does not allow for the elevator to be between floors, the analogous requirement in the ITL specification would be that the elevator is always at some floor. Interestingly enough, there is no requirement in the ITL specification that the elevator always be at some floor. It appears that the elevator's service policy can be violated if the elevator is initially not at any floor.

$\forall t :$

$\exists i : on\_floor(i, t)$

or  $\exists j : above\_floor(j, t)$

• Constraints on door movement:

a. Any time the doors are open and a request is pending for some other floor, the doors will begin to close within time *max\_open\_time* unless they are obstructed.

– ITL

$b \neq a \supset [(open(a) \wedge light(b, dir)) \Rightarrow open(a) \wedge light(b, dir) + max\_open\_time]$

$\Box (in\_service \wedge \neg obstructed(a)) \supset * closing(a)$

– State Predicate

$\forall t, i, j :$

$doors\_open(i, t)$

and  $i \neq j$

and  $request\_light(j, t)$

and  $\forall t_1 \in (t, t + max\_open\_time] : \neg doors\_closing(i, t)$

*implies*

$\exists t_2 \in (t, t + max\_open\_time] : doors\_obstructed(i, t)$



b. The states opening, open, closed, and closing are complete and mutually exclusive.

– ITL

$$(opening(a) \vee open(a) \vee closing(a) \vee closed(a)) \wedge ((opening(a) \vee open(a)) \Leftrightarrow \neg (closing(a) \vee closed(a))) \wedge ((opening(a) \vee closing(a)) \Leftrightarrow \neg (open(a) \vee closed(a)))$$

– State Predicate

$\forall t, i :$

Either:

$doors\_opening(i, t)$

or  $doors\_open(i, t)$

or  $doors\_closed(i, t)$

or  $doors\_closing(i, t)$

and  $doors\_opening(i, t)$  implies

$$\neg (doors\_open(i, t) \text{ or } doors\_closed(i, t) \text{ or } doors\_closing(i, t))$$

and  $doors\_open(i, t)$  implies

$$\neg (doors\_opening(i, t) \text{ or } doors\_closed(i, t) \text{ or } doors\_closing(i, t))$$

and  $doors\_closed(i, t)$  implies

$$\neg (doors\_open(i, t) \text{ or } doors\_opening(i, t) \text{ or } doors\_closing(i, t))$$

and  $doors\_closing(i, t)$  implies

$$\neg (doors\_open(i, t) \text{ or } doors\_closed(i, t) \text{ or } doors\_opening(i, t))$$

c. Between the time a set of doors is open and it begins closing, the doors remain open.

– ITL

$$open(a) \Rightarrow [before\ closing(a)] \Box open(a)$$

– State Predicate

$\forall t_1, t_2, i :$

$doors\_open(i, t_1)$

and  $doors\_closing(i, t_2)$

and  $\forall t_3 \in (t_1, t_2) : \neg doors\_closing(i, t_3)$

implies

$$\forall t_4 \in (t_1, t_2) : doors\_open(i, t_4)$$

d. Between the time a set of doors is closed and it begins opening, the doors remain closed.

– ITL

$$[closed(a) \Rightarrow before\ opening(a)] \Box closed(a)$$

- State Predicate

$\forall t_1, t_2, i :$

$doors\_closed(i, t_1)$

and  $doors\_opening(i, t_2)$

and  $\forall t_3 \in (t_1, t_2) : \neg doors\_opening(i, t_3)$

implies

$\forall t_4 \in (t_1, t_2) : doors\_closed(i, t_4)$

- e. When the elevator is not in motion, the doors for the floor that it is on are in the same state as the elevator doors while all other doors are closed.

- ITL

$opening(lift) \Leftrightarrow \exists a : 0 \leq a \leq n \wedge opening(a)$

$0 \leq a \leq n \supset opening(a) \Rightarrow closed(a) \sqcap atfloor(a) \wedge (opening(lift) \Leftrightarrow opening(a)) \wedge (open(lift) \Leftrightarrow open(a)) \wedge (closing(lift) \Leftrightarrow closing(a)) \wedge (closed(lift) \Leftrightarrow closed(a))$

These ITL assertions have the same intent as our assertion, but are different for two reasons. First, the ITL specification has no concept of movement. Second, an assertion of the form  $[I \Rightarrow J]P$  is defined to be vacuously true whenever there is no  $J$  interval following an  $I$  interval. Thus, if the doors begin opening and never subsequently become closed, these assertions place no restrictions on the behavior of the doors.

- State Predicate

$\forall t, i :$

$on\_floor(i, t)$

implies

$doors\_open(i, t) = lift\_open(t)$

and  $doors\_opening(i, t) = lift\_opening(t)$

and  $doors\_closed(i, t) = lift\_closed(t)$

and  $doors\_closing(i, t) = lift\_closing(t)$

and  $doors\_obstructed(i, t) = lift\_obstructed(t)$

and  $\forall j$  such that  $j \neq i : (doors\_closed(j, t) \text{ and } lift\_closed(t))$

- All doors are closed when the elevator is in motion.

There is no analogue to this assertion in the ITL specification.

$\forall t, i :$

$above\_floor(i, t)$

implies

$lift\_closed(t)$

and  $\forall j : \text{doors\_closed}(j, t)$

- The doors become open within time *opening\_time* of when they began opening.

\* ITL

$[\text{opening}(a) \Rightarrow \text{open}(a)] \square \text{in\_service} \supset < \text{opening\_time}$

This is an incorrect formalization of the assertion in ITL. As noted earlier, if the doors begin opening and never subsequently become open, then the above assertion is defined to be vacuously true. Thus, instead of saying that the doors must become open within time *opening\_time*, the ITL assertion actually requires that if the doors become open at some later point, then they must have become open within time *opening\_time*. Consequently, the ITL assertion does not even require that the doors become open at some later point. The correct way to formalize this assertion in ITL is:

$[\text{opening}(a) \Rightarrow \text{opening}(a) + \text{opening\_time}] \diamond (\neg \text{in\_service} \vee \text{open}(a))$

This says that within *opening\_time* of when the doors become opening, they either become open or the elevator becomes out of service.

\* State Predicate

$\forall t_1, i :$

$\text{doors\_opening}(i, t_1)$

implies

$\exists t_2 \in (t_1, t_1 + \text{opening\_time}) : \text{doors\_open}(i, t_2)$

- f. The doors become closed within time *closing\_time* of when they began closing unless they are obstructed in the meantime.

- ITL

$[\text{closing}(a) \Rightarrow \text{closed}(a)] \square (\text{in\_service} \wedge \neg \text{obstructed}(a)) \supset < \text{closing\_time}$

This has the same type of error as the ITL specification of the previous assertion.

- State Predicate

$\forall t_1, i :$

$\text{doors\_closing}(i, t_1)$

and  $\forall t \in (t_1, t_1 + \text{closing\_time}) : \neg \text{doors\_closed}(i, t)$

implies

$\exists t_2 \in (t_1, t_1 + \text{closing\_time}) : \text{doors\_obstructed}(i, t_2)$

- g. When the doors become obstructed, they begin opening within time *reaction\_time*.

- ITL

$[\text{obstructed}(a) \Rightarrow \text{opening}(a)] \square \text{in\_service} \supset < \text{reaction\_time}$

This contains the same type of flaw as the ITL specification for the previous assertion.

- State Predicate

$\forall t_1, i :$

$doors\_obstructed(i, t_1)$

*implies*

$\exists t_2 \in (t_1, t_1 + reaction\_time) : doors\_opening(i, t_2)$

- h. When the doors become open as a result of being obstructed, they begin closing within time *dwell\_time* of when they become open.

- ITL

$[(obstructed(a) \Rightarrow open(a)) \Rightarrow closing(a)] \square inservice \supseteq dwell\_time$

This contains the same type of flaw as the ITL specification for the previous assertion.

- State Predicate

$\forall t_1, t_2, i :$

$doors\_obstructed(i, t_1)$

and  $\forall t \in (t_1, t_2) : \neg doors\_open(i, t)$

and  $doors\_open(i, t_2)$

and  $t_1 < t_2$

*implies*

$\exists t_3 \in (t_2, t_2 + dwell\_time) : doors\_closing(i, t_3)$

- i. The time between when the doors become open at a floor and when they begin closing is at least *min\_open\_time*.

- ITL

$[open(a) \Rightarrow closing(a)] > min\_open\_time$

This contains the same type of flaw as the ITL specification for the previous assertion.

- State Predicate

$\forall t_1, t_2, t_3, i :$

$\forall t \in (t_1, t_2) : \neg doors\_open(i, t)$

and  $doors\_open(i, t_2)$

and  $doors\_closing(i, t_3)$

and  $t_1 < t_2 < t_3$

*implies*

$t_3 - t_1 > min\_open\_time$

j. The doors can only be obstructed when they are closing.

– ITL

$[obstructed(a) \Rightarrow ] closing(a)$

– State Predicate

$\forall t :$

$lift\_obstructed(t) \text{ implies } lift\_closing(t)$

and  $\forall i : doors\_obstructed(i, t) \text{ implies } doors\_closing(i, t)$

– When the elevator reaches a floor for which there is a pending request, the doors begin opening.

$t_1 < t_2$

and  $\forall t \in (t_1, t_2) : \neg on\_floor(i, t)$

and  $on\_floor(i, t_2)$

and  $request\_light(i, t_2)$

*implies*

$doors\_opening(i, t_2)$

• Constraints on change of direction:

a. The elevator can only change direction from down to up when there are no requests pending at a lower floor.

– ITL

$b < a \supset [before\ goingup \Rightarrow atfloor(a) \supset \neg light(b, dir)]^{10}$

– State Predicate

$\forall t_1, t_2, i, j :$

$\forall t_3 \in [t_1, t_2) \neg going\_up(t_3)$

and  $going\_up(t_2)$

and either:

$(on\_floor(i, t_2) \text{ and } j < i)$

and  $(above\_floor(i, t_2) \text{ and } j \leq i)$

*implies*

$\neg request\_light(j, t_2)$

---

<sup>10</sup>The predicate *goingup* is used in the ITL model to denote the event that the elevator has decided to change directions and move upwards.

- b. The elevator can only change direction from up to down when there are no requests pending at a higher floor.

– ITL

$$b > a \supset [before \neg goingup \Rightarrow atfloor(a) \supset \neg light(b, dir)]$$

– State Predicate

$$\forall t_1, t_2, i, j :$$

$$\forall t_3 \in [t_1, t_2) : going\_up(t_3)$$

$$\text{and } \neg going\_up(t_2)$$

and either:

$$(on\_floor(i, t_2) \text{ and } i < j)$$

$$\text{or } (above\_floor(i - 1, t_2) \text{ and } i \leq j)$$

*implies*

$$\neg request\_light(j, t_2)$$

- c. The elevator cannot reverse directions when it is between floors.

Since the ITL specification does not allow the elevator to be between floors, this assertion is not represented in the ITL specification.

$\forall t :$

$$reverse(t)$$

*implies*

$$\exists i : on\_floor(i, t)$$

**3.2.2.2 Elevator Service Policy** — We place the following requirements on the service provided by the elevator:

- ITL

$$[newrequest(a, dir) \Rightarrow newrequest(a, dir) + max\_service\_time] \Box (inservice \wedge \neg obstructed) \Leftrightarrow * open(a)$$

This asserts that in the interval of length *max\_service\_time* starting with a new request for service at floor *a*, the doors will open at least once at floor *a* if and only if no doors are obstructed. It is unreasonable to require that there were not any obstructions if the doors open within time. For example, *max\_service\_time* must be relatively large for a building with many floors. Then, if the elevator is at floor 0 when a request is made at floor 1, it is quite possible that the doors might be temporarily obstructed at floor 0 and still arrive at floor 1 within time. So, it seems that simply requiring that if the doors are not obstructed, then they will open at least once is a more reasonable policy. This is the policy that we use in our model.

- State Predicate

If no doors are obstructed in the meantime, the elevator will service a floor within time *max\_service\_time* of a request for service at that floor.

*up\_requested(i, t)* or *down\_requested(i, t)* or *floor\_requested(i, t)*

and  $\forall t_1 \in [t, t + \text{max\_service\_time}] : \neg \text{servicing}(i, t_1)$

*implies*

$\exists t_2 \in [t, t + \text{max\_service\_time}] : \text{lift\_obstructed}(t_2)$

**3.2.2.3 Analysis of Elevator**— The proof of the service policy for the elevator follows after showing that there exists an appropriate constant *M* such that the following lemmas hold:

- If the elevator is at floor *i* or between floor *i* - 1 and floor *i* and going down, then it will reach floor 0 within time *i* × *M* unless it reverses directions, a set of doors is obstructed, or all requests have been serviced in the meantime.

$\forall i, t_1 :$

*(on\_floor(i, t<sub>1</sub>) or above\_floor(i - 1, t<sub>1</sub>))*

and *¬going\_up(t<sub>1</sub>)*

and  $\forall t \in (t_1, t_1 + i \times M) : \neg \text{on\_floor}(0, t)$

*implies*

$\exists t \text{ in } (t_1, t_1 + i \times M) :$

*(reverse(t) or lift\_obstructed(t) or ( $\forall j : \neg \text{request\_light}(j, t)$ ))*

- If the elevator is at or above floor *i* and going up, then it will reach floor *MAX<sub>floor</sub>* within time (*MAX<sub>floor</sub>* - *i*) × *M* unless it reverses directions, a set of doors is obstructed, or all requests have been serviced in the meantime.

$\forall i, t_1 :$

*(on\_floor(i, t<sub>1</sub>) or above\_floor(i, t<sub>1</sub>))*

and *going\_up(t<sub>1</sub>)*

and  $\forall t \in (t_1, t_1 + (\text{MAX}_{\text{floor}} - i) \times M) : \neg \text{on\_floor}(\text{MAX}_{\text{floor}}, t)$

*implies*

$\exists t \in (t_1, t_1 + (\text{MAX}_{\text{floor}} - i) \times M) :$

*(reverse(t) or lift\_obstructed(t) or ( $\forall j : \neg \text{request\_light}(j, t)$ ))*

- c. If the elevator arrives on a floor for which there is a request pending, the doors become open within time  $M$ .

$\forall t_1, t_2, i :$

$\forall t \in (t_1, t_2) : \neg on\_floor(i, t)$

and  $on\_floor(i, t_2)$

and  $request\_light(i, t_2)$

implies

$\exists t_3 \in (t_2, t_2 + M) : doors\_open(i, t_3)$

Our goal is to show that:

If the doors are not obstructed during the interval of length  $max\_service\_time$  starting when a request is made, then the request is serviced at some time during that interval.

We will assume that:

$$2 \times M \times (MAX_{floor} + 1) < max\_service\_time.$$

The system parameters must be chosen so that lemmas a-c hold for some such value of  $M$  to validate this analysis.

Since we only consider times at most  $max\_service\_time$  time units after when service is requested in the following analysis, we can assume without loss of generality that the doors are not obstructed. Consequently, we ignore the possibility of the doors becoming obstructed in the following analysis.

The analysis begins by assuming that a request has been made at floor  $i$ . Suppose that the elevator is at or below floor  $j$  and traveling down when the request is made. If  $i$  is below  $j$ , then the elevator may not change directions until the request is serviced. Thus, lemma a ensures that within time  $i \times M$  the request is either serviced or the elevator reaches floor 0. In order for the elevator to reach floor 0, it previously must have passed through floor  $i$ . Then, lemma c ensures that the request is serviced within time  $i \times M + M$ . In either case, the request is serviced in time  $(i + 1) \times M$ .

Now, suppose that  $i$  is above  $j$ . Then, within time  $j \times M + M$  it either reverses direction or reaches the bottom floor and reverses direction (by lemma a). Then, analysis similar to that used in the previous case shows that the floor will be serviced within time  $(i + j + 2) \times M$ <sup>11</sup>. The analysis for the case in which the elevator is initially travelling upwards is similar. Since  $i + j < 2 \times MAX_{floor}$  and we are assuming that  $max\_service\_time > 2 \times M \times (MAX_{floor} + 1)$ , the service policy is satisfied.

Thus, the analysis is reduced to finding a value for  $M$  suitable for justifying the lemmas. It suffices to choose  $M$  such that it is greater than the maximum time between leaving a floor and leaving the next floor while a request is pending. So,  $M$  must be large enough to address the movement time between floors, the opening time for doors, the time doors remain open, and the closing time for doors. This means that  $M$  must be chosen so that it is larger than:

<sup>11</sup> Lemma b ensures the request will be serviced within time  $(i + 1) \times M$  from its arrival at floor 0



$$movement\_time + opening\_time + max\_open\_time + closing\_time$$

to satisfy lemmas a-c. Combining this with our earlier analysis, we see that the service policy holds whenever:

$$2 \times (movement\_time + opening\_time + max\_open\_time + closing\_time) \times (MAX_{floor} + 1) < max\_service\_time$$

### 3.2.3 Summary of the Example

In this section we have provided an example of a real-time system and policy. Although the system and policy are relatively simple, they are complex enough to provide a nontrivial example. The example demonstrates how a set of relatively simple conditions can be generated that imply the satisfaction of a real-time policy. Rather than directly showing the elevator's service policy is satisfied, it suffices to show that:

$$2 \times (movement\_time + opening\_time + max\_open\_time + closing\_time) \times (MAX_{floor} + 1) < max\_service\_time$$

and that the model of the elevator is consistent with the real elevator.

By providing the ITL specifications along with the state predicates, we make it possible to compare ITL with state predicates. Although ITL is more concise, it suffers from two disadvantages.

First, it is more difficult to write an ITL specification. The ITL specification in [17] contained several errors that were difficult to find due to the difficulty in humans interpreting ITL specifications. Simple ITL specifications are very understandable, but the more complex specifications are very difficult to decipher. While the state predicates are less concise, it is easier to determine the correctness of a state predicate model since it is easier to interpret the specifications.

Second, [17] does not describe any proof rules for ITL. Thus, it is not clear how an analysis such as that provided for the state predicate model could be performed on the ITL specification. [17] addresses this by making use of the decidability of the logic to argue that it is feasible to develop a tool that can automatically analyze an ITL model. The premise seems to be that such a tool would make proof rules unnecessary. It seems unlikely that a completely automated tool could analyze the ITL specification of a complex system in a reasonable amount of time. The current state of the art is such that totally automated theorem provers are very weak. This would mean that a human would need to develop a proof strategy to simplify the analysis. Since there currently are no proof rules, the human would have to translate the ITL specifications to a more primitive notation such as our state predicates. Then, any advantage of using ITL is lost. This disadvantage can be addressed if a nice set of proof rules can be developed for ITL.

Thus, the state predicate approach is the most useful of the approaches we considered for stating and proving real-time policies.

### 3.3 SUMMARY OF AVAILABILITY MODELS

In this section we have addressed fault tolerance, classes of denial of service threats, and provided a worked example of the specification and analysis of a real-time system.

The policies developed for fault tolerance and liveness are quite general. They both make use of the CSP formalism which we found to be the most useful for developing availability policies for distributed systems, with the exception of real-time policies. Our approach for using these policies to analyze a system is:

- a. Develop a CSP specification of the system ignoring the possibility of faults.
- b. Determine the set of faults to be tolerated and the effects of each fault.
- c. Extend the CSP specification of the system to address the possibility of faults.
- d. Demonstrate that the system satisfies our fault tolerance policy. Since this demonstrates that no new behaviors are introduced by faults, the possibility of faults can be ignored in any subsequent analysis.
- e. Demonstrate that the CSP specification that ignores the possibility of faults satisfies any of our availability policies that are desired.

As discussed previously, further research is needed to determine possible extensions to our policies that might be required to address nondeterminism. A complete worked example of the above approach must be developed, too. This would provide validation for the policies and the approach and allow any inadequacies to be addressed. Since a similar worked example must be done to validate the security policies developed in [4], Distributed Trusted Mach and Theta-DOS are the obvious candidates for the worked example. Although both of these systems are too complex to perform a complete analysis as a proof of concept, a representative set of servers could be analyzed to validate our approach.

The same must be done for our work with real-time policies. A moderately complex application with real-time constraints must be identified to serve as a more realistic worked example of the approach.

Armed with the availability policies developed in this section and the security policies developed in [4], we are now ready to consider making trade-offs between security and denial of service.

## SECTION 4

### TRADE-OFFS

In [5] a detailed analysis was made of possible ways to deny service in a distributed system. As part of the analysis, known techniques for preventing these denial of service attacks were examined, and, for each technique, the possible security implications were discussed. In this section we summarize some of the basic trade-offs that necessarily occur when attempting to provide both MLS security and a high degree of assured service. We also examine some approaches to security policy issues that can be used when confronted with these trade-offs.

The definition given in [5] for denial of service is that the system does not satisfy its specifications. Such specifications are usually functional and stated in terms of specific actions, responses and response times that the system must provide. On the other hand, the security policy for a system is usually a higher level policy that is stated in terms of information flow in a more abstract manner[7], and the implications of such a policy are often not immediately obvious. This is necessary because the concept of illegal information flow that the definition is attempting to capture is subtle and not easily recognized from functional specifications.

In an MLS distributed system, the challenge is to satisfy both the denial of service policy and the security policy. There are areas in which the policies are mutually supportive. For example, techniques aimed at protecting the integrity of data on the system help guarantee that neither service is denied nor security is compromised based on corrupted data. Unfortunately, other techniques used to ensure that one policy is satisfied may often lead to the second policy being violated. Some of the most common reasons for this are:

- **Shared Finite Resources.** In a system in which processes share system resources, the potential exists for service denial based on one process accumulating and holding these system resources. A number of techniques have been developed to prevent such an occurrence. However, most of these techniques involve some type of approach that, in an MLS system, could be used to signal information from a high level to a low level subject.
- **System Algorithms.** Algorithms that control system-wide processing, e.g. scheduling algorithms or communication protocols, often involve communication among processes that, in an MLS system, may be at different levels. The danger exists that this processing can be manipulated to signal information from high to low.
- **Shared Data Structures.** Techniques for both sharing system resources and performing system-wide processing often rely on shared data structures for maintaining control information. If these data structures are global, then they present a potential security problem.

The above conflicts between assuring service and maintaining a secure system lead to a number of trade-offs that may be necessary. These trade-offs could result in either the security policy or the assured service policy being modified.

#### **Security Policy Trade-offs.**

To provide the degree of service that the system requires, it may be necessary to make modifications to the security policy that the system is designed to enforce. There are a number of possible ways in which this could be done.

- **Trusted Subjects.** To provide assured service, it may be necessary, or desirable, to use a technique that requires that information be written at several levels or that communication occur across levels. If ordinary subjects were allowed to do this, then they would have the potential to easily pass information from a high to a low level. By encapsulating the processing in a special subject, which is identified as a trusted subject, and then analyzing the processing that this subject does to ensure that its special privileges cannot be misused, it is sometimes possible to minimize the security risk of the operation. Trusted subjects typically represent the exceptions to the systems access control policy.
- **Allow Covert Channels.** In certain systems, the dangers that a covert channel poses may be minimal compared to the effort required to exploit the channel. If it is known that covert channels are not a major concern of the system, then an access control policy, such as the Bell and LaPadula policy [1], that is not concerned with covert channels could be used. The danger in this approach is that since there is no attempt to identify covert channels, there is no real way of knowing whether the assumption that such channels are not a major threat to the security of the system is valid.
- **Allow Restricted Covert Channels.** In most systems it may not be possible, or desirable, to eliminate all downward information flows. Hence, a strict information flow policy that says that there is no flow from high to low would not be provable. In such cases, it may be possible to modify the definition of noninterference so that the particular information flow is removed from the policy statement. Using this modified information flow policy it is then possible to continue the search for other unauthorized information flows. Three techniques that may be useful are:

Conditional noninterference.

Effectively ignoring channels.

Probabilistic noninterference.

**Conditional Noninterference.** In [7] the notion of Conditional TH-Guardedness was introduced to provide an information flow policy that allowed well-identified exceptions to the policy. Such a conditional noninterference policy states that except for certain *privileged* operations there is no information flow from a high to a low level. These privileged operations can then be analyzed individually to determine the risk involved with each. While some of the operations trusted to violate a system's access control policy might be included in this set of privileged operations, [4] demonstrates that not all trusted operations downgrade information. It also is possible that some of the privileged operations are not trusted operations. So, although there is some similarity between the notion of trusted subjects and subjects privileged to violate noninterference, they are independent concepts.

**Effectively Ignoring a Channel.** A well-known potential covert channel involves the identifier that is generated when a new object or subject is created on the system [10]. If a low level process creates an object, waits, then creates a second object, by comparing the ids of the objects created, the process can determine if any other objects or subjects were created in between. So by either creating an object or not between the time when the low level objects are created, a high level subject can signal to the low level subject. While this is an obvious covert channel, it may not be of much interest on the system if, for example, the identifiers are encrypted so that the order in which they are generated is not recognizable.

It is possible to abstract out from the information flow analysis known covert channels present in the system that are not of interest. This approach has been called *effectively ignoring* the channel [20]. The idea is that when a high level operation occurs, its effects should not be visible to any lower level subject except for the results of the particular known channel. That is, it should be possible to effectively ignore the operation at the lower level except for the change made through the channel. This allows one to define an information flow policy that can still be proven even in the presence of known covert channels. The technique clearly identifies these channels and, at the same time, allows the information flow analysis to continue on the remainder of the system. The technique differs from conditional noninterference in that in the later the entire operation is excluded from the noninterference analysis, while in the former, only that part of the operation that results in the channel is excluded. For the identifier example, the high level operations that create new objects and subjects can be effectively ignored except for the effect that they have on the generation of future identifiers. This allows the remainder of these operations to still be part of the noninterference analysis.

**Stochastic Policies.** In [7] we defined a stochastic information flow policy that required:

For any sequence of inputs and low-level outputs, the probability that the system will generate the given sequence of low-level outputs from the sequence of inputs is the same as the probability the system will generate the low-level output sequence from only the low-level inputs of the given input sequence.

Illegal information flow would occur if the above described probabilities were not the same indicating that some high-level event affected the low-level outputs.

In [15] Gray presents a different version of such a stochastic policy, called P-restrictiveness, and suggests that by altering the probabilities that certain events occur in a system, it may be possible to satisfy a stochastic information flow policy while at the same time limiting certain denial of service attacks. In particular, the example that is discussed is the classic reader-writer problem in which a data object is being written by a low-level process and read by a high-level process.

Since readers and writers cannot both access the data object at the same time, some protocol must be used to synchronize the accesses. To prevent downward information flow, if a low-level writer requests to write the data object, then any high-level readers that are currently accessing the object are preempted. This presents the possibility that high-level readers can be permanently denied access to the data by the actions of low-level writers.

Gray proposes that this denial of service problem may be partially addressed by controlling the probabilities with which certain events can occur in the system. In the example, by

requiring that specific inputs be polled for, it is possible to guarantee that the write request from the low-level process only be serviced a certain percentage of the time, say  $p$ . (If a request is not serviced, it may be serviced at a later time. In effect, servicing of the request is delayed.) When the low-level write request is finally serviced, of course, all current high level readers are preempted, but the idea is that, in the meantime, the high-level process may be given a longer time to complete its read. The intent is that the high-level process will not be preempted as often.

Although the stochastic information flow policy is relatively new and needs further development, as mentioned in [7], there are a number of serious difficulties with it. Moreover, there appear to be a number of difficulties in attempting to use such a policy to prevent denial of service also.

- a. Determining the appropriate probabilities to assign to each event is a difficult process. As the example in [15] shows, unless the probabilities are determined in exactly the right manner, the stochastic information flow policy will not be satisfied.
- b. The system must then enforce that the appropriate events occur with the required probabilities. This may complicate the implementation considerably.
- c. The approach suggested has the effect of slowing the entire system down. Events can only occur with a given probability. If the system is in a state in which there is only one possible event, and if that event can only occur with probability  $p$ , then  $1-p$  of the time, the system will have to wait.
- d. In the end, the approach may not prevent denial of service. The reader-writer problem is basically a problem in concurrent access. As such, it is time dependent. Depending on the length that the read request takes, it may be possible for a malicious low-level writer to still interrupt it before it is finished most of the time.

The last two points indicate that the effect of juggling the probabilities with which certain events on the system can occur is basically equivalent to slowing down the frequency with which the low-level process can request a write and, thereby, pose a denial of service threat. An alternate approach that would be simpler, have the same effect, and avoid the probabilistic complications, would be to just put a time delay on any write request. When the write request is made, delay a fixed time before actually servicing the request. This would allow any high-level reads a fixed amount of time to complete before they are preempted. Of course, just as with the probabilistic approach, reads that did not finish in time would be preempted.

- **Provide a Heterogeneous Policy.** In a distributed system composed of heterogeneous nodes, it may not be possible to prove a policy that holds for all nodes, or, alternatively, the policy that is proven may only be as strong as the weakest node. An alternate approach might be to prove the strongest policy that is possible for each node, and then identify the manner in which the weaker nodes can compromise the system and attempt to address these threats on an individual basis.

#### **Assured Service Trade-offs.**

If it is determined that a potential covert channel should be eliminated from the system because it poses too great a threat to the security of the system, then a number of possible approaches may be taken depending on which is most appropriate. All of these approaches will, to one degree or another, affect the level of service that the system can provide.

- **Restrict Access to Shared Resources.** For resources that are abundant enough that they can be partitioned among levels, it may be possible to statically allocate quotas of the resource among the levels so that the sum of the quotas never exceeds the total amount of the resource. This eliminates the channel at the expense of disallowing all sharing of the resource. Such an approach was used by Multics in allocating directory space.

A variation of this strategy would involve allowing higher level processes to use lower level quotas that were currently unused. When a lower level process desired the use of the resources, however, the system would have to preempt them from the higher level process.

- **Preempt High Level Processes.** For shared resources, this approach is similar to the variation mentioned above except that there are no quotas at any level. The resource being shared is allocated on a first come, first serve basis. However, to prevent a high level process from signalling a low level process by using all the resource, whenever a low level process requests the resource and there is none available, a high level process that has the resource allocated may have to relinquish it. This prevents the covert channel at the expense of causing the high level process to face possible starvation because it cannot retain control of the resource long enough to finish its processing.
- **Per Level Algorithms.** Rather than implement system-wide algorithms that require access to information at multiple levels, it may be possible to modify the algorithms so that the processing can be done on a level by level basis with minimal communication across levels. For example, it should be possible to implement recovery algorithms on a level by level basis.
- **Use Stale Data.** For data that is being shared across levels, that is, read by higher level processes and written by low level processes, the concern is that the higher level processes access consistent data. If a low level process is in the act of updating data, then the high level process must wait until the update has been completed or risk getting data that is inconsistent. This again presents the possibility that the high level processes can be indefinitely postponed. One method of avoiding this would be to keep older, consistent, versions of the data available for the high level processes to read. This allows the high level processes to complete their processing but at the expense of using data that may be out of date. For a real-time  $C^2$  application, however, this may not be appropriate. Another possibility would be to inform the application that the data is stale and let it decide whether to roll back.
- **Audit and Regulate.** In many cases, it may not be feasible to completely eliminate a potential information flow from high to low. However, it may be possible to identify when such a flow can occur and audit it. If the audit mechanism is sophisticated enough, it may also be possible to identify when the channel is being exploited and react appropriately. Regulating the degree to which the channel can be used might also be done via time delays.

This approach has two drawbacks.

- a. It results in certain parts of the system being slowed down which, in practice, might limit service availability.
- b. It can only be used on operations that can be audited. Operations that occur so frequently that it would be infeasible to audit them, such as disk accesses, could not be handled in this manner.

In summary then, trade-offs between security and assured service seem inevitable. A security analysis that uses conditional noninterference, augmented by analyzing trusted subjects and by effectively ignoring certain aspects of operations, would seem to give the most complete analysis while allowing enough flexibility to make compromises to assured service when necessary. Channels can be isolated and identified and, if necessary, audited and regulated. In some cases, heterogeneous policies may be necessary. The trade-offs for assured service are less clear and usually must be dealt with on a per situation basis. In particular, this might involve changing specifications to allow for some time delays.

To demonstrate the usefulness of the approaches for performing trade-offs, an example should be worked of a reasonably complex system that has been analyzed with respect to both availability and security policies. Due to the generality of the approaches for performing trade-offs, this would provide very useful guidance to developers and evaluators of distributed, secure systems.



## SECTION 5

### CONCLUSION

In this report we have:

- Discussed various formalisms that might be used for specifying and analyzing  $C^2$  systems.
- Described an approach for analyzing such systems with respect to fault tolerance.
- Developed policies to address classes of denial of service threats including starvation, deadlock, mutual starvation, and likelock.
- Provided an example of a real-time system and policy.
- Discussed trade-offs that can be made between security and denial of service.

Of the formalisms we have considered, CSP seem the most appropriate for use with  $C^2$  systems. Unfortunately, it does not appear reasonable to add real-time to CSP. Real Time Logic (RTL) and Interval Temporal Logic (ITL) seem the most appropriate for real-time systems. Due to the difficulty we had in developing an RTL model of the elevator system, we feel that ITL is more usable than RTL. However, our state predicate approach that uses only the state predicate portions of RTL appears to be much more useful than ITL.

We have found that the most useful approach for analyzing systems with respect to availability is:

- a. Develop a CSP specification of the system ignoring the possibility of faults.
- b. Determine the set of faults to be tolerated and the effects of each fault.
- c. Extend the CSP specification of the system to address the possibility of faults.
- d. Demonstrate that the system satisfies our fault tolerance policy. Since this demonstrates that no new behaviors are introduced by faults, the possibility of faults can be ignored in any subsequent analysis.
- e. Demonstrate that the CSP specification that ignores the possibility of faults satisfies any of our availability policies that are desired.

This provides an argument that the system satisfies the availability policies even in the presence of the classes of faults addressed by the fault tolerance mechanism.

Since it is not reasonable to incorporate real-time into CSP, this approach must be modified for systems with real-time policies. The two main changes are:

- a. A formalism such as timed state predicates should be used rather than CSP.

- b. The effect of fault tolerance mechanisms on the behavior of the system must be considered in the specifications of the system.

Due to the application specific nature of real-time policies, we worked an example of the analysis of a real-time system with regard to a real-time policy rather than attempting to define a general policy. Although the system we analyzed is relatively simple, the same approach can be used for more complex systems.

We found that the most reasonable way to address trade-offs between secrecy and availability is to weaken the respective policies to obtain a policy that clearly:

- a. Identifies the conflicts between secrecy and integrity.
- b. Identifies the degree of secrecy and integrity that holds in each case of conflict.
- c. Identifies the absolute policies that hold when there are no conflicts.

For example, conditional noninterference can be used to require that there be no violation of the secrecy policy except for certain well-defined exceptions. A complete policy can be obtained by extending the policy to define the permissible actions the system can take for each of the exceptions. Other ways of weakening policies to remove conflicts include stochastic policies and the concept of effectively ignoring.

In order to demonstrate the usefulness of the policies and approaches described in this report, it is necessary to apply them to more realistic systems. By providing a worked example of the analysis of a moderately complex system, the policies and approaches can be validated. In addition, due to the generality of the policies and approaches, it should be straightforward for system developers and analysts to use the worked example as a guide to follow in the analysis of their systems. Since the system analyzed should be a secure, distributed system, Distributed Trusted Mach and Theta-DOS are the most reasonable candidates for the worked example. Although both systems are too complex to analyze merely as a proof of concept, a representative portion of either system could be analyzed as the worked example.

## References—

- [1] D.E. Bell and L.J. LaPadula. Secure computer system : Unified exposition and multics interpretation. Technical Report MTR-2997, July 1975.
- [2] Flaviu Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, pages 57–78, February 1991.
- [3] Edward A. Schneider, D.G. Weber, and Tanja de Groot. Specification/verification of temporal properties for distributed systems: Issues and approaches. Technical Report RADC-TR-89-376, Rome Air Development Center, February 1990.
- [4] Mike Endrizzi, Todd Fine, and Tom Haigh. Security in distributed  $c^2$  systems. Technical report, Secure Computing Technology Corporation, 1991.
- [5] Mike Endrizzi, Todd Fine, Tom Haigh, and Sudhakar Yalamanchili. Security and denial of service in distributed systems. Technical report, Secure Computing Technology Corporation, 1990.
- [6] Farnam Jahanian and Aloysius Ka-Lau Mok. Safety analysis of timing properties in real-time systems. *IEEE Transactions on Software Engineering*, pages 890–904, September 1986.
- [7] Todd Fine, Richard O'Brien, and Bill Wood. Availability in distributed  $c^2$  systems. Technical report, Secure Computing Technology Corporation, 1991.
- [8] Dov Gabbay, Amir Pnueli, Saharon Shelah, and Jonathan Stavi. On the temporal analysis of fairness. In *Seventh Annual Symposium on Principles of Programming Languages*, pages 163–173. ACM, 1980.
- [9] Morrie Gasser. *Building a Secure Computer System*. Van Nostrand Reinhold Company, New York, first edition, 1988.
- [10] J.T. Haigh and W.D. Young. Extending the noninterference version of mls for sat. *IEEE Transaction on Software Engineering*, pages 141–150, Feb 1985.
- [11] Maurice P. Herlihy. A quorum-consensus replication method for abstract data types. *ACM Transactions on Computing Systems*, February 1986.
- [12] Maurice P. Herlihy and Jeannette M. Wing. Specifying graceful degradation. *IEEE Transactions on Parallel and Distributed Systems*, pages 93–104, January 1991.
- [13] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.
- [14] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Publishing Company, 1979.
- [15] James W. Gray, III. Probabilistic interference. In *1990 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 170–179, 1990.
- [16] Richard Alan Karp. Proving failure-free properties of concurrent systems using temporal logic. *ACM Transactions on Programming Languages and Systems*, 6(2):239–253, April 1984.

- [17] L. Lamport, P. Melliar-Smith, L. Moser, I. Greenberg, and J. Rushby. Algorithms for fault tolerant distributed systems. Technical Report RADC-TR-89-162, Rome Air Development Center, October 1989.
- [18] Susan Owicki and Leslie Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):155-495, July 1982.
- [19] James L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1981.
- [20] Todd Fine and J. Thomas Haigh and Richard C. Obrien and Dana L. Toups. Noninterference and unwinding for lock. In *The Computer Security Foundations Workshop II*, pages 22-28, 1989.
- [21] Jeffrey D. Ullman. *Principles of Database Systems*. Computer Software Engineering Series. Computer Science Press, Rockville, MD, second edition edition, 1982.
- [22] D.G. Weber. Formal specification of fault-tolerance and its relation to computer security. *Communications of the ACM*, pages 273-277, 1989.

**MISSION  
OF  
ROME LABORATORY**

*Rome Laboratory plans and executes an interdisciplinary program in research, development, test, and technology transition in support of Air Force Command, Control, Communications and Intelligence (C<sup>3</sup>I) activities for all Air Force platforms. It also executes selected acquisition programs in several areas of expertise. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C<sup>3</sup>I systems. In addition, Rome Laboratory's technology supports other AFSC Product Divisions, the Air Force user community, and other DOD and non-DOD agencies. Rome Laboratory maintains technical competence and research programs in areas including, but not limited to, communications, command and control, battle management, intelligence information processing, computational sciences and software producibility, wide area surveillance/sensors, signal processing, solid state sciences, photonics, electromagnetic technology, superconductivity, and electronic reliability/maintainability and testability.*